AD-A201 117

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
N/A

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-88-138

6a. NAME OF PERFORMING ORGANIZATION
University of Massachusetts

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
Rome Air Development Center (COTD)

6c. ADDRESS (City, State, and ZIP Code)
Department of Computer and Information Science
Amherst MA 01003

7b. ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Rome Air Development Center

8b. OFFICE SYMBOL (If applicable)
COTD

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F30602-81-C-0206

8c. ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
62702F	5581	21	PA

11. TITLE (Include Security Classification)
LOCAL/GLOBAL CONTROL INTEGRATION

12. PERSONAL AUTHOR(S)
John A. Stankovic, Don Towsley, C.G. Rommel, Spiridon Pulidas, Ravi Mirachandaney

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM Jul 81 TO Jul 87

14. DATE OF REPORT (Year, Month, Day)
August 1988

15. PAGE COUNT
284

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	07	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Decentralized Control
Distributed Algorithms
Scheduling

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The goal of this contract was to investigate decentralized control with respect to the scheduling and reallocation functions of distributed computer systems. We have made significant progress on developing and analyzing several scheduling and reallocation algorithms. In particular, we have investigated distributed scheduling algorithms where tasks are independent of each other and the subnet imposes a non-negligible delay on task transfers. We have also studied distributed scheduling algorithms that specifically consider collections of related tasks which we classify as distributed groups and clusters. Since many distributed systems have nodes which are multiprocessors, we have also addressed multiprocessor scheduling. We have studied scheduling in such an environment by analytically analyzing the performance of fork-join jobs and by developing a multi-class, multiprocessor scheduling algorithm. During this contract period, we have also developed a decentralized reallocation algorithm, analyzed it via simulation, and in the analysis, emphasized different forms and costs of cooperation. We show that decentralized reallocation is significantly better (over)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
□ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas F. Lawrence

22b. TELEPHONE (Include Area Code)
(315) 330-2158

22c. OFFICE SYMBOL
RADC (COTD)

Previous editions are obsolete.

Block 19. Abstract (Cont'd)

than centralized reallocation. Finally, we also developed decentralized estimation techniques to be used in conjunction with distributed scheduling algorithms.

TITLE: LOCAL/GLOBAL CONTROL INTEGRATION

Principal Investigators: John A. Stankovic
Don Towsley

University of Massachusetts
Department of Computer and Information Science
Amherst, MA 01003

REFERENCE: subcontract number RI-44896X
OGCA No. 86A538

1 Introduction

The goal of this contract was to investigate decentralized control with respect to the scheduling and reallocation functions of distributed computer systems. We have made significant progress on developing and analyzing several scheduling and reallocation algorithms. In particular, we have investigated distributed scheduling algorithms where tasks are independent of each other and the subnet imposes a non-negligible delay on task transfers. We have also studied distributed scheduling algorithms that specifically consider collections of related tasks which we classify as distributed groups and clusters. Since many distributed systems have nodes which are multiprocessors, we have also addressed multiprocessor scheduling. We have studied scheduling in such an environment by analytically analyzing the performance of fork-join jobs and by developing a multi-class, multiprocessor scheduling algorithm. During this contract period we have also developed a decentralized reallocation algorithm, analyzed it via simulation and in the analysis emphasized different forms and costs of cooperation. We show that decentralized reallocation is significantly better than centralized reallocation. Finally, we also developed decentralized estimation techniques to be used in conjunction with distributed scheduling algorithms.

This report discusses the main results of our work and includes 7 reports and papers as Appendices. All of these were generated during this contract. We divide the report into five main sections: distributed load sharing in the presence of delays, multiprocessor scheduling, decentralized reallocation, the design of efficient parameter estimators for decentralized load balancing policies, and a summary.

2 Distributed Load Sharing in the Presence of Delays

2.1 Independent Jobs and Delay

Adaptive Load Sharing in the Presence of Delays:

The primary focus of one part of our research has been on the following aspects of adaptive load sharing:

- We have developed and studied analytical models for three simple load sharing algorithms called Forward, Reverse and Symmetric, under the assumption that task transfers over the network experience non-negligible delays. Various performance studies have been conducted using these models, the details of which are presented in [1]. For example, we have seen that load sharing provides improvements even at very high delays ($\geq 80 - 100$ times the average service time of tasks), when the loads are very high ($\geq 90\%$). At lower loads however, the algorithms can only tolerate smaller delays and still provide benefits from load sharing. Further, it is seen that remote state information ceases to provide any benefits when delays are greater than 2-3 times the service time.

- In the above work, we had made some simplifying assumptions to facilitate our analysis. In order to study the validity of some of these assumptions, we conducted an in-depth study of Receiver-Initiated load sharing algorithms. The details of this study are presented in [2]. From this study, we have been able to determine that our earlier assumptions were valid under a fairly large range of system parameters. For instance, we had assumed that probes experienced no delays while tasks experienced large delays. It was seen from the model results that as long as probes experienced a small fraction of the delays compared to tasks, the model behaved like one with zero delays for probes. In light of the fact that in most systems, tasks are likely to be several orders of magnitude greater than probes, this was a reasonable assumption.

- All our work until this point had assumed a homogeneous system of nodes, where both the task arrival rate and the average service time were the same at each node. In many real systems the identical arrival rate assumption does not hold (we call these Type-1 Heterogeneous systems) and in others, the processing speeds of the nodes may be different (although the nodes may have the same processing capability). These are referred to as Type-2 systems. From our studies regarding load sharing in such systems, we have been able to observe various interesting facts: For example, in Type-1 systems, it was seen that Random assignment performed very well in comparison to probing, particularly Reverse, when the systems were highly heterogeneous. In less heterogeneous systems, probing seemed to be more effective. The benefits of load sharing were more pronounced as the systems became less balanced, for a fixed system load. Also, the algorithms could tolerate higher delays. In Type-2 systems the performance of probing algorithms was much better than the Random assignment algorithm and here too, higher delays were tolerated than the homogeneous counterparts. Further, it was seen that the choice of thresholds appeared to be more critical in case of highly heterogeneous Type-2 systems, with incorrect thresholds producing very high response times.

- Most work on load sharing is performed under the assumption of time constant (albeit probabilistic) arrival rates. We know that this is not true in most real systems, where the arrival rates can vary drastically between different times of the day, days of the week and so on. Stated simply, we have looked at estimation techniques to determine the changing loads in the systems and subsequently determine high level policies which will enable the load sharing algorithms to adapt to these changes.

  From our studies regarding the estimation of changing parameters (e.g., the system load, task arrival rate), we have seen that simple techniques are able to estimate these parameters quite effectively. From our earlier studies, we know that small (5-10%) errors in estimating the parameters are not likely to have a major impact on performance. Consequently, once the changed loads (or arrival rates) are estimated, simple policies are used to adapt the internal parameters in response to these changes. It is seen that this type of adaptive load sharing policy performs only slightly worse (3-5%) than the optimal strategy (one that uses the optimal parameters and control at all times).

2.2 Distributed Groups/Clusters

Scheduling Distributed Programs: In the current literature, the development of distributed programming techniques is expanding. These techniques are designed to take advantage of the parallelism in distributed systems by partitioning the program into tasks. During this contract period we developed preliminary algorithms for scheduling distributed programs in a distributed system in which a processor-sharing scheduling algorithm is used. Each program is composed of a set of tasks which may be executed at any system site, and individual tasks of the program may communicate with each other. Tasks may move between any site before and during execution. During this contract we have taxonimized distributed programs, developed an understanding of the specific new issues that must be dealt with in scheduling distributed programs, and developed preliminary algorithms for scheduling distributed programs which deal with these issues. Simulation programs are being constructed and the details of the algorithms will be modified as the results of the simulations suggest. In the remainder of this section we will provide some basic definitions related to distributed programs, highlight the *special problems* to be addressed, and give a brief overview of the approach we are undertaking.

We begin with the following definitions. A **distributed system** is a system composed of two or more physically separate sites connected by a communication system referred to as a subnet. Classes of programs which may execute on a distributed system include:

- **Sequential Program:** Sequential programs are programs composed logic that executes as a single non-parallel thread. Such a program cannot take advantage of the inherent parallelism of the distributed system by separating itself into multiple tasks. It can take advantage of the distributed system by moving perhaps to a lightly loaded site.
- **Parallel Program:** This is a program composed of tasks which can execute at the same time. Examples of these programs are found in many numerical programs such as matrix computation and non-numerical programs such as the partitioned traveling-salesperson problem or graphics programs such as the histographing program.
- **A Distributed Program:** These are programs composed of tasks which are physically separated. Both sequential and parallel programs might be distributed. Examples of these programs which are distributed but need not be parallel, are programs which are pipelined but the tasks of the program reside on separate hosts, and subroutines with remote sends/waits. However, most distributed programs are parallel programs.

The above definitions serve only to classify programs. For scheduling it is more important to consider the communication aspects of parallel and distributed programs. We define:

- **Highly Autonomous Program:** A highly autonomous program is a program composed of tasks which have little or no intra-task communication. They are usually characterized by an initialization, execution, and cleanup phases. The initialization phase is used in the determination of the tasks and their assignments. The execution phase is the computation phase. This program is assumed to have little or no communication between tasks. The cleanup phase reports the results and terminates the program. An example of this class of program is the **fork-join** program. This type of program is conducive to analytical analysis as described elsewhere in this report.

- **Autonomous with Asynchronous Communication Program:** The autonomous with asynchronous communication program is characterized by asynchronous communication. Typically, this program also includes an initiation, execution, and cleanup phase. Further, communication may be in various amounts. In many cases the communication does not significantly interfere with the execution of the program.
- **Autonomous with Synchronous Communication Program:** The autonomous with synchronous communication program is characterized by tasks which communicate with sends/waits. Like the programs above, these programs have an initialization, execution, and cleanup phase. These programs could easily exhibit a slow down if distributed.

Two key parameters of parallel and distributed programs are the amount and type of communication. We use these key paramters to define the types of parallel and distributed programs on interest to us in this study. Consequently, we define:

- **A Cluster:** A cluster is a parallel program composed of a collection of tasks which communicate either frequently, in large amounts, or have many sends/waits synchronization points. In general, distributing this program is not beneficial to its execution.
- **A Distributed Group:** This is a program which is composed of a collection of tasks that interact, but potentially benefit from executing at different sites.

2.2.1 Problems Associated with Distributed Groups and Clusters

In scheduling distributed groups and clusters we must account for transfer delays, overheads, different speeds of the different hosts in the system, have some knowledge of the number of tasks and types of communication of a given distributed program, and how and what information to collect on the state of the system and its various programs. These are typical scheduling problems but they must now be addressed with respect to distributed programs. Rather than discussing these issues, let us concentrate on the new problems that arise with respect to scheduling parallel and distributed programs.

The main issues for distributed groups are:

- For distributed groups two opposing forces exist. From the nature of distributed groups, we should physically separate the tasks due to the potential benefits. Conflicting with this desire is the general state of hosts on the system. For example, if one site is lightly loaded and all other sites are overloaded, it would then seem to be reasonable to assign tasks to the lightly loaded site even if two or more of them were from the same distributed group. In another instance, if multiple members of a distributed group are created at the same site, then it would seem reasonable to keep one group member at the site the group was created, if the load at this site is moderate, and disperse the others to other system sites being careful not to send multiple members to the same site.

There are several special problems for clusters. We may classify these aspects as the cycle problem, the cluster collection problem, the null site problem, the acceptable service problem, the reaction delay problem, and the idle site problem.

- **Cycle Problem:** If we accept that a good algorithm seeks to collect clusters, then a cyclic problem can occur. In this problem several sites move cluster tasks in a cyclic fashion about the system such that little is achieved and the tasks are receiving significant transfer delays.

  An example of the cyclic problem is the following. Suppose that there are three tasks, $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ of a cluster at sites $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$. To obtain better performance , site $\mathcal{X}$ would move task $\mathcal{A}$ to site $\mathcal{Y}$; site $\mathcal{Y}$ would move task $\mathcal{B}$ to site $\mathcal{Z}$; and site $\mathcal{Z}$ would move task $\mathcal{C}$ to site $\mathcal{X}$. In most cases, the net effect would not be a performance increase, but rather a performance degradation introduced by the reassignment.

- **Cluster Collection Problem:** If the system wishes to collect a cluster at a site, then there is a possibility that the system would be unable to find an existing site available for efficient execution of the cluster. The reason for this is that we are now trying to assign all the cluster tasks to a site which probably has other tasks and the net load on this site may now be too high. This problem requires that we disperse one or more of the existing non-cluster tasks so that we can accept the collection of the tasks of the cluster and not have too busy a site. This is not a simple problem because we must not disperse already collected tasks of other clusters at the site, nor disperse a distributed group task to a site with other distributed group tasks.

- **Null Site Problem:** This problem is related to the situation in which a site is very lightly loaded but has no tasks of a cluster. The cluster assumption is that cluster tasks would perform better at the same site. However, some algorithms might exclude from consideration the site with no tasks. We must avoid such algorithms.

- **Acceptable Service Problem:** Suppose that several tasks of a cluster are receiving acceptable service at a site; however, some are not receiving acceptable service. The problem is whether we consider the entire cluster as candidates for reassignment or only those which are receiving unacceptable service. Our approach is to treat the entire cluster as a unit any time that one member if being processed.

- **Reaction Delay Problem:** The scheduling algorithm must minimize delay between recognizing that the cluster exists and collecting the cluster to improve performance. The longer it takes to perform these steps, the less benefit there will be to clustering. We call this delay the **reaction delay**.

- **Idle Site Problem:** The idle site problem is characterized by a site being without work despite the fact that work exists within the system. It is reasonable to expect that the idle site should receive a complete cluster if this situation occurs. In our scheme, idle sites become active participants in trying to move an entire, but curently dispersed cluster to itself. If this is not possible, they will try to move distributed group members, simple programs and already collected clusters to itself if it is estimated that it would be beneficial to do so.

General approach to scheduling distributed groups and clusters:

- The main ingredient of our approach is to compute the **average time to complete all tasks at a site,** ATC_{site}. Our approach attempts to predict the impact of adding or deleting a task (where the task could be a simple task, one or more cluster tasks, or distributed group tasks) on both the local and receiving site. We also consider transfer delays in the prediction of the response time. We predict transfer delays by using an estimate of the task size in kilobytes and the current transfer rate across the subnet.

  We assume

  - Each site knows or can easily find the location of all distributed group and cluster tasks.
  - When scheduling one must treat the tasks of a cluster as one entity. Our scheduling algorithm works by having sites with tasks of a given cluster negotiate with each other.

  As an example of a scheduling algorithm logic, consider each of the special cluster problems listed above.

  - **Cycle Solution:** Cycles can occur in two different ways. One, it might be possible for two hosts concurrently executing the scheduling algorithm to simultaneously decide to transmit cluster members and cause a cycle. This problem is solved by choosing a unique coordinator to decide when and where to move tasks of a given cluster at any point in time. The second possibility is that two non-overlapping executions of the scheduling algorithms (possibly widely separated in time) re-migrate tasks in a cycle. This problem is not as severe, for indeed, if widely separated in time, then a cycle might be appropriate. However, intuitively, too many such cycles are probably indications of race conditions and should be avoided. Consequently, the algorithm constrains the task not to move more than a fixed number of times.
  - **Cluster Collection Solution:** To solve this problem sites act as either sinks or coordinators for clusters. When a site goes idle, it becomes a sink for clusters. As a sink it seeks to bring clusters to itself. Sinks also attempt to acquire distributed group tasks and simple programs. Cluster coordinators perform negotiation to enable

clusters to collect at a site, yet not overload that site. The details of the coordination are under investigation.

- **Null Site Solution:** A null site is a site with no elements of a particular cluster. To consider null sites we allow any site to bid on the tasks of a cluster at a site. This is done by broadcasting a request for the cluster. The coordinator also considers null sites when making its decisions.
- **Acceptable Service Solution:** The solution of this problem is to start bidding on a cluster when any one task is executing poorly. Such bidding may be initiated on any individual task exceeding a threshold but may lead to the exchange of a partial cluster, bargaining on several partial clusters, or an assignment of simple programs, distributed groups, and partial clusters.
- **Reaction Delay Solution:** As soon as a task enters the system, it is considered for assignment. If the task is recognized as a cluster member. we initiate the coordinator to immediately try to collect the cluster, and if not, to find the best site for the cluster member.
- **Idle Site Solution:** The idle site problem is solved by having the idle site become **an active sink** for dispersed clusters. As an active sink, it seeks to bring complete clusters to itself. This solves two problems. First, the idle site problem is solved by bringing the work to the site. Secondly, it can solve the null site problem by allowing a site with no cluster tasks to request the cluster.

In this section we described some of the issues and problems associated with scheduling distributed programs on distributed systems. Again, this work is preliminary, and algorithms are currently being developed and evaluated by simulation.

3 Multiprocessor Scheduling

3.1 Fork–Join Jobs

Our work in scheduling fork–join programs includes:

- scheduling fork–join programs at a single site with processor-sharing, and
- scheduling fork-join programs on a multiprocessor.

We have completed papers for each of these above two cases. The first paper is entitled "Analysis of Fork–Join Jobs Using Processor-Sharing." (See [3]). A fork–join job is a special type of parallel program composed of a set of tasks each of which can be scheduled independently of the other, but the job is not considered complete until all the tasks complete. In [3], we derive an expression for the mean response time of a fork-join job in a single server processor-sharing queueing system. We also derive an expression for the mean response time of a fork-join job conditioned on the required service time of the largest task. Each task service time is assume to be an exponentially distributed random variable. We provid both lower and upper bounds on mean response time of fork-join jobs. The lower bound to the mean response time of the fork-join problem is very tight when the number of tasks in the job is large (≥ 7) and/or the server utilization is high. Numerical results are developed that provide various insights such as that fact that processor-sharing scheduling at the job level is better than at the task level.

We have also completed a second paper entitled "A Comparison of the Processor Sharing and First Come First Serve Policies for Scheduling Fork-Join Jobs in Multiprocessors." (See [4]) This paper is an extension of the work done in [3], using different analysis techniques.

In this second paper, a model of a shared memory multiprocessor that executes *fork-join* parallel programs as a bulk arrival $M^X/M/c$ queueing system is developed. Here a fork-join job is one that consists of a set of X tasks. All of the tasks arrive simultaneously to the system and the job is assumed to complete when the last task completes. We develop tight upper and lower bounds for the mean response time of such programs when the scheduling discipline is processor sharing under the assumptions of exponential task service times and a Poisson job arrival process. We study two processor sharing policies, one called *task scheduling* processor sharing and the other called *job scheduling* processor sharing. The first policy schedules tasks independently of each other and allows parallel execution, whereas the second policy schedules entire jobs as a unit and thereby does not allow parallel execution of an individual program. We find that the job scheduling policy exhibits better performance than task scheduling only on systems with a small number of processors, where the system is operating at high loads and is executing programs that can sustain a large degree of parallelism. Consequently, in general, task scheduling outperforms job scheduling. We also compare the performance of the processor sharing policy with first come first serve. We find that first come first serve exhibits better performance over a wide range of systems. The paper also studies the performance of processor sharing and first come first serve with two classes of jobs, and when a specific number of processors is statically assigned to each of these classes. The most interesting conclusion is that static partitioning of processors to classes of jobs must be avoided because it gives very poor performance. The next section describes a multiprocessing algorithm that does dynamic partitioning.

3.2 A Multi–Class Multiprocessor Scheduling Algorithm

In this section we outline the ideas that we have developed as the basis for new multiprocessor/distributed (MD) scheduling algorithms. Each node of the distributed system contains a local scheduler that is based on the new type of multiprocessor scheduling algorithm that we have developed. Local schedulers at the different nodes are similar in structure, but differ as a function of the size (number of processors) and expected functionality of that particular node. Each of the nodes of the network coordinate with each other via the global scheduling algorithm. We hypothesize that both the local and global schedulers cleanly separate policy and mechanism.

Our multiprocessor scheduling algorithm is based on the assumption that there are four classes of jobs. They are:

- short jobs composed of a small number of tasks, called class S,
- long jobs composed of a small number of tasks, called class L (these are cpu hogs),
- jobs composed of a large number of parallel tasks, called class M, where M stands for multiple tasks, and
- jobs which require a dedicated portion of the multiprocessor, called class D.

Our goal is to make the algorithm flexible by giving different weights to each of the four classes of jobs as a function of the size of the multiprocessor, and the state of the system. The breakdown of work into the four classes is to ensure no loss of service to jobs which require short bursts of cpu time, but at the same time allow for CPU hogs, and for cooperative jobs that require a large number of processes (e.g., possibly some AI-like component of a system). Our algorithm also permits dedicated service (assignment of processors) to particular jobs. Our analytical work described in [5] clearly shows that static partitioning of a multiprocessor is a bad idea. Hence, this algorithm dynamically partitions the processors in an effort to avoid idle processors.

Outline of the Algorithm

There are one or more ready queues for each class (but the remaining discussion assumes a single queue for each class). Processors, when idle, will extract work from one of these queues. The algorithm will be described in a piecemeal fashion driven by the following questions. This approach allows us to choose alternative answers to the main questions, resulting in new algorithms. Since we have not yet completed this work, we only describe the basic algorithm here.

The major questions that must be dealt with include:

- What is a job, what is a task, how are they related, what are the dynamics between jobs and tasks, and between tasks and tasks?
- How does a task (or job) get into a given queue? Will tasks (or jobs) switch between queues?

- How do one or more processors get assigned to a job? When a job needs more than one procesor how does it collect those processors?
- What happens when a task must wait for a resource or for another task?
- What are the important system parameters needed in order to have a flexible and efficient algorithm?

The initial answers to these questions are stated next in terms of an algorithm (albeit spread out over several pages). Follow on work is necessary to evaluate this algorithm.

A job is defined as the initial unit of work requested by a user. A Job Control Block is created to represent a job. When a job is activated it is assigned one or more tasks (the active processing entity). Tasks may spawn other tasks. Tasks are the dispatchable entities and when ready to execute they are placed in one (and only one) of the ready queues. Tasks are represented by Task Control Blocks and they point to the Job Control Block or to a parent Task Control Block. Jobs may move between queues which implies that all of its tasks move with it. An interesting alternative is to treat each task of a job independently. However, a job's response time will be most affected by the slowest task, and if tasks of that job requiring little cpu time finish fast by being in the S queue, it would probably have little affect on the overall job response time, but slow down other truly short jobs. A job is complete when all of its tasks are complete.

When a job is activated it may request a particular class, S, L, M, or D, or it may not request anything in particular. If a class is requested, then the job proceeds to that queue, else it proceeds to the S queue.

Then,

- if the job receives more than t sec move it to the L queue,
- if the job creates more than m tasks then move it to the M queue.

NOTES: (i) jobs never go back to S or L from M, or back to S from L.
(ii) jobs get into D queue only if requested.

Let N be the total number of processors, then N(S) is the number assigned to S class jobs, N(L) the number assigned to class L jobs, and N(M) the number assigned to M class jobs. $N(S) + N(L) + N(M) = N$. Further, let N(D) be the maximum number of processors assigned to class D jobs. $N(D) \leq N$, so class D jobs borrow processors from the other classes. Our major motivation for having class D jobs borrow processors is that we believe that we should discourage class D jobs because they will impact all other jobs considerably, and setting aside some fixed number of processors for them will waste resources. Given the type of multiprocessors of interest to us (all processors on one shared bus) it makes no difference which set of processors is assigned to each class. A main idea of our algorithm is that although there may be an initial partitioning (by system parameter settings) of the processors among the classes, processors will be allowed to move between classes as a function of the load. This approach attempts to provide a certain amount of minimum performance for each class, but when a given situation demands it, larger numbers of processors can be assigned to the overloaded class.

For example, a class S processor when idle, goes to the class S ready queue to obtain its next task. If there are no ready tasks, and the number of idle processors in class S is greater than TH(S), then this processor checks the class M queue. If no work exists there, then the processor checks the class L queue. If there is still no work, then it returns to the class S queue and waits for new work. If this processor remains idle for greater than T sec and the number of free processors in class S is still greater than TH(S) then this processor again checks the other queues. This is not particularly expensive since the processor would just be idle anyway. Similar thresholds exist for the L and M classes, i.e., TH(L) and TH(M). The processors in the other classes operate in a similar manner, i.e., they can temporarily move themselves into other classes.

However, there may be some congestion as multiple processors try to simultaneously access a given queue. Appropriate synchronization mechanisms are required. For multiprocessors in the 5-100 range, we do not expect serious contention on the 4 class ready queues.

When processors service tasks in another queue, an important question is how long does a processor remain servicing tasks in that other queue. There seem to be at least three choices. One, a processor only serves one task; when that is complete, it returns to its original class and will only serve tasks in other queues again if the same criteria are met. Two, the processor stays in the new class until the original class idle processors goes below some threshold. Three, it remains in the new class; this dynamically adjusts the size of the old class and the new class. However, option 3 seems to require that at least some minimum number of processors remain in each class, e.g., MIN(S), MIN(L), and MIN(M). Consequently, each queue (S, L and M) has a minimum and original number of processors. Those processors between the minimum and the initial (current) allocation are free to dynamically reallocate themselves to other queues based on current load. This means that no particular processors are assigned as floaters (as found in the isarithmic congestion control scheme where certain messages float around the system).

The class D queue is treated differently. A number of parameters exist with respect to this queue. First, the D class can be turned ON_D or OFF_D. If ON_D, then the first job in the queue is chosen; the requested number of dedicated processors must be less than N(D). D queue jobs must borrow processors from other classes. Initially, the job will wait for time TD for enough idle processors to be simultaneously available. TD can be set to zero. If enough idle processors are available within time TD then it dedicates these to the job. In practice, an idle processor of the S class periodically checks the D queue. If there is work then it is its responsibility to handle this D class job, e.g., by watching for the proper number of idle processors. A global variable that records the number of idle processors must be maintained. If time TD expires and enough processors have become available then perform the following, depending on the priority of the task D job:

Priority 1: Get the required number of processors as quickly as possible by taking all free processors, and the next q free processors until a sufficient number is acquired. Notice this means that many processors may be idle while waiting for the number requested.

Priority 2: Simply assign processors to the D class job: when all assigned, then interrupt all of those involved and give them to the D class job. Assigning could mean that we only assign currently free processors and those as they become free processors (but while waiting, some may subsequently become busy), or just arbitrarily assign processors and interrupt them.

Priority 3: Proportionally and temporarily reduce the size of each of the classes S, L, and M based on their current sizes. Here again we have to decide whether to wait or interrupt.

Note that the dispatching discipline for each queue is round robin with the quantum being short in the S class, being long in the L class, and being moderate in the M class. However, other policies can easily be substituted within each class. When a task blocks for a resource or for another task, it is assigned to a wait queue. When the condition for which it was waiting occurs, it is placed back in the same ready queue unless the job has moved in the meantime, in which case the task goes to its new queue. Currently, tasks do not have a priority. Again, it is not difficult to use priority scheduling within a given queue rather than round robin.

The main research work still needed to be performed includes:

- Futher developing the local multiprocessing scheduling algorithm described above and variations to it.
- Developing coordination algorithms that can serve as the global scheduling algorithm.
- Implementing the local and global algorithms on the two Sequent Balance multiprocessors. Comparing variation of the algorithms to each other, and also comparing the new local scheduling algorithm to the current Sequent scheduling algorithm.
- Modelling various types of distributed computations.
- Validating the models on the multiprocessors.
- Integrating the notion of deadlines and other real-time constraints into the algorithms.

4 Decentralized Reallocation

We have developed efficient decentralized algorithms that reallocate real–time tasks after a processor in a distributed system fails [6]. Through extensive simulations, we showed that the decentralized algorithms perform significantly better than centralized reallocation algorithms. We tested the algorithms under different loads, different load patterns, different cost models for the reallocation algorithms themselves, different computation time requirements for the tasks, different external arrival rates while the reallocation is being processed, different quality of the state information, and for different size networks. In all these tests the decentralized algorithms consistently outperformed the centralized algorithms.

The main characteristics of our decentralized algorithm are that it is decentralized and reliable, it specifically considers deadlines of tasks, it attempts to utilize all the nodes of the distributed system to achieve its objectives, it is fast, it handles tasks in priority order, and it separates policy and mechanism. We have also determined that for reallocation in hard real–time systems it is more important to make quick decisions with out of date information rather than having higher cooperation which results in more accurate data but longer delays. These longer delays result in an overall performance detriment to the system. For a full description of the algorithm and its analysis see [6].

5 Design of Efficient Parameter Estimators For Decentralized Load Balancing Policies:

We consider distributed computer systems consisting of multiple host computers interconnected by a communication network. Jobs arrive at each host computer according to some arrival process and can be processed either locally or at other hosts after being transfered through the communication network. The results of jobs processed remotely are returned to the origin host computer. Communication delays are incurred due to the transfer of remote jobs and their results. In such an environment, *load balancing* attempts to improve the performance of the distributed system (e.g. to minimize the mean response time of a job) by using the processing power of the entire system to smooth-out periods of high congestion at individual nodes.This is done by transferring jobs from heavily loaded nodes to lightly loaded nodes.

Throughout this work we consider a class of *threshold* load balancing policies, where jobs at each host are divided into two classes; local and remote jobs [7]. *Local jobs* are those processed at the site of origination and *remote jobs* are those transfered for processing to another node. A job arriving to a node from the external world is processed locally only if the current queue length is less than a *threshold* parameter. Otherwise, the job is sent for remote processing to another host selected according to some probability distribution. Remote jobs are always accepted by the destination hosts and therefore jobs can move at most once. Both local and remote jobs are processed according to a first-come-first-served (FCFS) discipline at a given host.

Obviously, such a threshold policy has control parameters (e.g. threshold values and transfer probabilities for every host computer), that require fine-tuning in order to yield optimal or near-optimal performance. An efficient distributed algorithm for determining the optimum values for parameters present in the decentralized threshold scheduling policy has been developed. The algorithm is iterative in nature and at each iteration load balancing parameters at each host are updated. This algorithm requires that each host computer be able to estimate the change in throughput and expected queue length due to a change in either the threshold or the job arrival rate. The host computers exchange this gradient information with each other and use these quantities to update the load balancing parameters for the next iteration of the algorithm.

Our approach towards estimating the gradients with respect to the threshold relies on the assumption that either the arrival process or the service time are exponentially distributed [7]. By adding a small perturbation to the observed parameter (i.e. a small decrease in the threshold), the estimator we have developed determines the effect of the change on the performance metric of interest (i.e. throughput, expected queue length), taking advantage of the *memoryless property* of the arrival process or the service time distribution. The estimator is originally presented for a system with Poisson arrivals and then is modified, so that it can be applied to a system with general distribution of arrival times but exponentially distributed service time. The arrival rate is estimated during the execution of the algorithm, while the increase in running time and the memory requirements (i.e. number of counters required) are very low. The major advantage of this technique is its *on-line* potential, since it effectively attempts to provide performance sensitivity information while the actual system is running. The performance of the proposed estimation technique has been investigated through simulations. As it turns out from the simulation results, the estimation algorithm converges very fast even for short observation periods (i.e. 200 job completions).

A different method is proposed to obtain estimates with respect to the arrival rate. The method is based on a class of theorems derived from Likelihood Ratios and is extremely well suited to regenerative systems. A busy cycle of the processor where the estimator runs has been used as the regeneration period. By evaluating the above estimation technique through simulations, we realized that the observation period must be long enough (e.g. 2000 busy cycles) in order for the estimator to give consistent estimates of the desired gradient. However, the method can be effectively applied to systems involved in the decentralized threshold scheduling policy discussed in [7], where we only need to compare derivatives without worrying about their absolute value.

Both of the proposed estimation techniques have been imbedded in the updating threshold algorithm and simulation results have been produced for a system with two host computers modeled as single server queueing systems [7]. It turns out that after a finite number of algorithm iterations, the behavior of the system, in a static environment is confined in a neighborhood of the optimal performance. A great improvement in the performance measure (average response time of a job) has been observed in the system executing the distributed load balancing algorithm compared to a system with no load balancing at all. Although we used short observation periods (50 or 100 busy cycles) in most of our experiments, the performance of the algorithm was not affected. It appears that accuracy in gradient estimation is not very crucial, since it is the *relative* values of the gradients, not their *absolute* values, which are more significant. Since the optimal thresholds are usually small values (e.g. less than six), it takes more iterations for the algorithm to converge if the initial thresholds are large. A heuristic modification has been made in order to improve the performance of the technique estimating gradients with respect to threshold, when the initial threshold values are large enough so that the nominal system does not even reach its threshold. It is interesting to notice that the overhead of exchanging gradients between the hosts of the distributed system is small since they are not an istantaneous information such as queue length. An interesting problem, which remains to be studied through simulations, is the *adaptivity* of the algorithm in a dynamically changing system environment. In such a case, when there is an imbalance in incremental delays due to a change in the system environment, we expect that the algorithm corrects the imbalance properly so that the system performance may be improved.

6 Summary

We have addressed all the issues presented in the original proposal and extended the work to also include multiprocessors. We have developed algorithms for :

- distributed scheduling with independent jobs,
- distributed scheduling of groups and clusters,
- scheduling fork-join jobs,
- multiple classes running on a multiprocessor,
- decentralized reallocation, and
- estimation techniques.

We also performed analytical analysis of distributed scheduling algorithms subject to significant task transfer delays, of fork–join jobs, and in conjunction with estimation techniques. We have used simulation in studying distributed groups and clusters, decentralized reallocation, and estimation techniques.

A sample of our major conclusions are listed below. The statements made are in the context of the assumptions found in the full papers presented in the appendix.

- delays are very important in scheduling and too much previous analytical work ignores these delays,
- if the delay in transferring a job is less than or equal to the service time then the particular scheduling algorithm is important and if the delay is greater than 10 times the service time then the algorithm does not play a significant role,
- simple threshold policies are suitable for independent jobs,
- the forward scheduling algorithm is better than the reverse scheduling algorithm at most loads and delays,
- the symmetric scheduling algorithm is significantly better the the forward and reverse algorithms and still a good load sharing algorithm even if delays are at 100 times the service time,
- the effect of delays due to probing is negligible,
- rarely are there any multiple outstanding probes,
- there seems to be no benefit from having different thresholds at the sender and receiver,
- distributed programs cannot be treated as independent jobs so more sophisticated algorithms are necessary when task characteristics are complicated,
- we have formulas for mean response time of fork-join jobs executing on multiprocessors as well as mean response time for fork-join jobs conditioned on largest service time,
- scheduling at the job level is better than at the task level on a single processor,
- scheduling at the task level is better than at the job level on a multiprocessor, but FCFS is better than both,
- do not statically partition the processors of a multiprocessor into fixed sized classes,
- decentralized reallocation is better than centralized reallocation,
- making quick decisions with old data is better than making slower decisions with more accurate data, for reallocation in a hard real–time system,
- global preemption is not necessary as part of the reallocation algorithm,
- On-line estimation of gradient information is necessary for determining the optimum values of parameters involved in decentralized load balancing policies.

- Accuracy in gradient estimation is not very crucial, since it is the relative values of the gradients, nor their absolute values, which are more significant in the proposed decentralized load balancing algorithm. Therefore, very short observation intervals can be used for estimating the desired gradient information.
- A great improvement in the performance measure (average response time of a job) has been observed in the system executing the distributed load balancing algorithm compared to a system with no load balancing at all.
- Since optimal thresholds are usually small values (e.g. less than six), it takes more iterations for the algorithm to converge if the initial thresholds are large.
- The overhead of exchanging gradients between the hosts of the distributed system is small since they are not an instantaneous information (such as queue length).

7 References

[1] Mirchandaney, R., D. Towsley, and J. Stankovic, "Analysis of the Effects of Delays on Load Sharing," submitted to *IEEE Transactions on Computers.*

[2] Mirchandaney, R., D. Towsley, and J. Stankovic, "Effects of Delays on Receiver-Initiated Load Sharing Policies," submitted to *Journal of Parallel and Distributed Computing.*

[3] Towsley, D. and R. Mirchandaney, "The Effects of Communication Delays on the Performance of Load Balancing Policies in Distributed Systems," *Proc. of the 2nd Int. Workshop on Applied Math and Performance/Reliability Models in Computer/Communications*, May 1987.

[4] Rommel, G., D. Towsley, and J. Stankovic, "Analysis of Fork-Join Jobs Using Processor Sharing," submitted to *Operations Research.*

[5] Towsley, D., G. Rommel, and J. Stankovic, "A Comparison of the Processor Sharing and First Come First Serve Policies for Scheduling Fork-Join Jobs in Multiprocessors, submitted to Symposium on High Speed Computer Systems, July 1987.

[6] Stankovic, J., "Decentralized Decision Making For Task Reallocation in a Hard Real Time System," accepted for publication in *IEEE Transactions on Computers.*

[7] Spirus Pulidas and Don Towsley, "Design of Efficient Parameter Estimators for Decentralized Load Balancing Policies," - UMASS Technical Report, TR 87-79.

[8] Mirchandaney, R., Adaptive Load Sharing in the Presence of Delays, PhD thesis, Univ. of Mass., 1987.

8 Appendix

This appendix contains copies of all the papers and reports generated from this contract. Some of these papers were also submitted with earlier quarterly reports.

The Effect of Communication Delays on the Performance of Load Balancing Policies in Distributed Systems[1]

Don Towsley and Ravi Mirchandaney
University of Massachusetts
Amherst, MA 01003
USA

Abstract

This paper is concerned with understanding the effects that communication delays have on the performance of load balancing policies in distributed computer systems. Two problems are addressed. First, how do load balancing policies behave as a function of communication delay? Second, what is the value of state information to load balancing policies that operate in systems with high communication delays? In response to the first question, a simple mathematical model is developed to study load balancing policies that rely on local state information. The conditions for the optimal policy in this model are developed. These conditions are then used to determine the behavior of the optimal policy as a function of communication delay.

The second question is addressed by analyzing a simple forward probing load balancing policy. A simple approximate model is developed for a system in which the transfer of state information from one node to another suffers nonnegligible communication delays. We find that whereas the use of local state information in a load balancing policy can significantly improve performance when communication delays are high (5-100 times the processing delay), the use of remote state information provides little or no further improvement.

1. INTRODUCTION

Job scheduling in distributed computer systems has received widespread attention in recent years. Numerous simulation and analytical studies have focussed on the design and analysis of various scheduling policies as in [LIVN82], [WANG85], [EAGE86] and [LEE87], etc. These studies usually focus on specific policies and attempt to determine how they perform with respect to each other. The studies

[1] This work has been supported by the National Science Foundation under grant ECS-8406402 and Rome Air Development Center under contract RI-44896X

also focus on the effects that different parameters have on their performance. Although many of these studies include communications delay as a parameter, they rarely study the effects of communication delay on the behavior of the algorithms. In many cases the communication delay is assumed to be small in comparison to processing times.

This paper focusses primarily on the effects of communication delay on the performance of load balancing (lb) policies in distributed computer systems. We are interested in studying the effect that increasing the communication delay will have on different classes of lb policies. We attempt to answer questions such as: What effect does increasing communication delay have on network usage incurred by lb policies? What is the effect on the performance of a lb policy? In the case that a lb policy requires a node to use remote state information, what effect will out of date state have on performance?

We provide a mathematical treatment of these questions where the model assumptions are motivated by empirical observations. We will examine closely two classes of lb policies. The lb policies in the first class require processors to make decisions based solely on local state information. Policies in the second class require nodes to probe other nodes in order to obtain information with which to make a decision. One of the results in this paper is that for the case of homogeneous distributed computer systems, remote state information is not useful whenever the communication delay is greater than or equal to three times the processing time.

The study of the first class of policies is based on work reported in [TANT85]. This earlier paper developed and analyzed a simple model of a static lb policy in a distributed computer system. We extend this model and the results to lb policies that use local state information. This model and the attendant results is found in Section 2.

We are unable to perform a general study of the second class of lb policies. Instead we develop a simple performance model for a forward probing lb policy operating in a homogeneous system. We show that network traffic decreases and that performance degrades as the communication delay increases. We further determine the value of communication delay at which state information becomes useless. This analysis is found in Section 3.

2. LOAD BALANCING POLICIES USING LOCAL STATE INFORMATION

We consider a distributed computer system as shown in Figure 1. The system con-

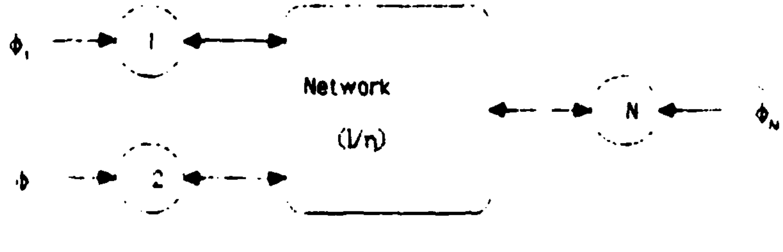


Figure 1: System Model.

sists of N nodes, which represent host computers, connected in an arbitrary fashion by a communications network. Each node consists of one processor contended for by jobs processed at the node. The processors may have different speed characteristics. However, they have the same processing capabilities, that is, a job may be processed on any node of the system.

Jobs arrive at node i, $i = 1, 2, \cdots, N$, according to a time invariant Poisson process with rate ϕ_i. The total job arrival rate is denoted by Φ. A job arriving at node i (referred to as the origin node) may be either processed at node i or transferred through the communications network to another node j (processing node). We assume that a job is never transferred from one node to another once it begins processing at a node and that it is never transferred more than once from one node to another. After the job is processed at node j, a response is returned to the origin node. The job service time is an exponential random variable (r.v.) with parameter μ_i at node i.

Most lb policies satisfying the above assumptions divide jobs into two classes, local and remote. A *local* job is one that is processed at its site of origin A *remote* job is one that is processed at some site other than its origin site. The flow of these two classes of jobs at node i is illustrated in Figure 2. Here β_i denotes the flow of completed local jobs and $\gamma_{j,i}$ denotes the flow of completed remote jobs at node i that originated at node j.

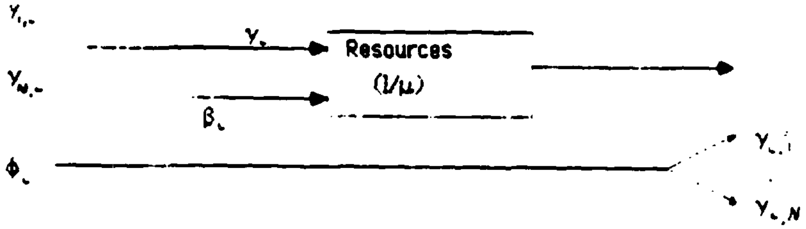


Figure 2: Node Model.

The response time of a job in the system consists of a node delay (queueing and processing delay) at the processing node in addition to a possible communication delay incurred due to job transfer. We denote the mean node delay of a local job and a remote job processed at node i by $F_i^{(\ell)}(\beta_i,\gamma_i)$ and $F_i^{(r)}(\beta_i,\gamma_i)$ respectively. Although local and remote jobs may obtain the same performance under some load balancing policies (i.e., static probabilistic lb), this may not always be true. We assume that the delay functions $F_i^{(\ell)}(\beta_i,\gamma_i)$ and $F_i^{(r)}(\beta_i,\gamma_i)$ at node i are differentiable, increasing and convex.

We assume that the mean communication delay for a job is $1/\eta$ and that it is independent of the origin and destination node as well as the load placed on the network.

We denote the mean response time of a job by $D(\beta,\gamma)$ where $B = [\beta_i]$ and $\Gamma = [\gamma_{i,j}]$. We find it useful to introduce $\gamma_i = \sum_{j=1}^{N} \gamma_{j,i}$, and $\gamma = \sum_{i=1}^{N} \gamma_i$. We can write the mean response time of a job as the sum of the mean node delay and mean communication delay; that is,

$$D(B,\Gamma) = \sum_{i=1}^{N} \frac{\beta_i}{\Phi} F_i^{(\ell)}(\beta_i,\gamma_i) + \sum_{i=1}^{N} \frac{\gamma_i}{\Phi} F_i^{(r)}(\beta_i,\gamma_i) + \sum_{i=1}^{N} \frac{\gamma_i}{\Phi\eta}. \tag{1}$$

Our goal is to balance the load, which is represented by the local job flows β_i and the remote job flow $\gamma_{i,j}$, in order to minimize the mean response time. The problem we are interested in then is

$$\begin{array}{lll} \text{minimize} & D(B,\Gamma) & \\ & & \\ \text{subject to} & \sum_{i=1}^{N}(\beta_i+\gamma_i)=\Phi, & \\ & 0\le\beta_i\le\phi_i, & 1\le i\le N, \\ & 0\le\gamma_i & 1\le i\le N. \end{array} \qquad (2)$$

We make several observations. First, there are only $2N$ unknowns, namely β_i and γ_i, $i=1,\cdots,N$. Once these quantities are obtained the flows $\gamma_{i,j}$ $(1\le i,j\le N)$ can be assigned any values so long as the overall flow of remote jobs into each node is unaffected. One assignment is

$$\gamma_{i,j}=\gamma_i(\phi_j-\beta_j)/\sum_{k\neq i}(\phi_k-\beta_k),\quad i=1,\cdots,N \qquad (3)$$

where flow is assigned from node j to node i in proportion to the total flow of jobs out of j.

Second, because of the assumption that $F_i^{(\ell)}(\beta_i,\gamma_i)$ and $F_i^{(r)}(\beta_i,\gamma_i)$ are convex, problem (2) is a convex nonlinear programming problem. We examine properties of this problem at the end of this section. Before that, we consider several specific load balancing policies that can be modeled by problem (2).

2.1. Static Probabilistic Load Balancing

The simplest load balancing policy that satisfies the above assumptions is the static lb policy studied by Tantawi and Towsley [TANT85]. We review some of the results of this study because they will be applicable to our study of threshold lb policies.

In this case, $F_i^{(\ell)}(\beta_i,\gamma_i)=F_i^{(r)}(\beta_i,\gamma_i)$ and the problem reduces to

$$\begin{array}{lll} \text{minimize} & D(\Theta)=\sum_{i=1}^{N}F_i(\theta_i)/\Phi+\gamma/(\Phi\eta), & \\ & & \\ \text{subject to} & \sum_{i=1}^{N}(\theta_i)=\Phi, & \\ & 0\le\theta_i\le\phi_i, & 1\le i\le N, \end{array} \qquad (4)$$

where $\theta_i=\beta_i+\gamma_i$.

There exists an optimal solution to this problem that partitions the nodes into three mutually exclusive sets of *sources*, *sinks*, and *neutrals*. Here a source node is one that only processes a fraction of its jobs locally and transfers the remainder to other nodes, a sink node is one that processes all of its jobs as well as some remote jobs and a neutral node is one that is neither a source nor a sink. Source nodes are further divided into sets of *idle* and *active* sources where an idle source processes no jobs and an active source processes some jobs. The partition of nodes into sources, sinks, and neutrals corresponding to the optimal solution can be obtained using a simple algorithm that sorts nodes according to their incremental node delays.

One provable property of this system of interest to us is that the flow of transfer jobs is a nonincreasing function of the network delay. Consequently, the effectiveness of the load balancing policy decreases with increasing communication delays.

2.2. Threshold load balancing

Numerous load balancing policies exist that require a node to decide to process a job locally or remotely according to the number of jobs at the processor. We describe two variations.

Threshold on local jobs only (SLO): Under this policy, node i processes a job locally if the number of local jobs does not exceed a threshold T_i. In addition, local jobs are given preemptive priority over remote jobs. Last, a job can not be transferred more than once. This policy was proposed by Lee [LEE87]. If the assumption is made that all arrivals are Poisson and that service times are exponential r.v.'s, then $F_i^{(\ell)}(\beta_i, \gamma_i)$ is

$$F_i^{(\ell)}(\beta_i,\gamma_i) = u_i\left\{1-(T_i+1)u_i^{T_i}+T_iu_i^{T_i+1}\right\}/\left[\beta_i(1-u_i)(u_i^{T_i+1})\right], \tag{5}$$

where $u_i = \phi_i/\mu_i$. The expression for $F_i^{(r)}(\beta_i, \gamma_i)$ for this system is

$$F_i^{(r)}(\beta_i,\gamma_i) = v_i/P_0 + \frac{\gamma_i B(C^2+1)P_0P_1 + 2v_I^2/P_0}{2(P_0-v_i)} \tag{6}$$

where

$$\begin{aligned}
v_i &= \gamma_i/\mu_i, & (7)\\
P_0 &= (1-u_i)/(1-u_i^{T_i+1}), & (8)\\
B &= (1-u_i^{T_i})_i/(1-u_i^{T_i+1}), & (9)\\
C &= \left[2(\mu_i-\phi_i u_i^{2T_i})/(\mu_i-\dot{\varphi}_i)^3 - 2(1+2T_i)u_i^{T_i}/(\mu_i-\phi_i)^2\right]/B^2 - 1. & (10)
\end{aligned}$$

At first glance, it is not clear how this can be transformed into the problem described above. However, if we ignore the integer constraint on the threshold T_i, then the problem can be transformed into one where we choose the local flow rate instead. This is done by substituting the following expression for T

$$T_i = [\ln(\phi_i - \beta_i) - \ln(\phi_i - \beta_i u_i)]/\ln u_i. \tag{11}$$

We are now left with justifying the Poisson assumption for remote jobs and convexity of $F_i^{(\ell)}(\beta_i, \gamma_i)$ and $F_i^{(r)}(\beta_i, \gamma_i)$. It has been shown in [LEE87] that under certain conditions, the arrival process of remote jobs approaches the Poisson process as $N \rightarrow \infty$. One such example is a system consisting of M clusters each containing $N_m \geq 2$ identical nodes (identical $\phi_i' s$, node delay functions) when $N_m \rightarrow \infty$, $m = 1, \cdots, M$ such that N_m/N_i is unchanged, $1 \leq i, m \leq M$. We have been unable to prove convexity of $F_i^{(r)}(\beta_i, \gamma_i)$. However all numerical evidence points to this conjecture being true.

Threshold on all jobs policy: Here the node processes a job locally if the number of jobs in the queue is below a threshold T_i. Otherwise it transfers the job to node j with probability $p_{i,j}$. Once a job is transferred, it is never transferred again. This policy was first described and studied by Eager et.al.[EAGE86]. Under the assumption that the processor is not required to transfer jobs from one node to another, the following expressions can be derived for $F_i^{(\ell)}(\beta_i, \gamma_i)$ and $F_i^{(r)}(\beta_i, \gamma_i)$

$$F_i^{(\ell)}(\beta_i, \gamma_i) = \frac{1 + T_i(u_i + v_i)^{T_i+1} - (T_i + 1)(u_i + v_i)^{T_i}}{(1 - u_i - v_i)(1 - (u_i + v_i)^{T_i})}, \tag{12}$$

$$\begin{aligned} F_i^{(r)}(\beta_i, \gamma_i) &= P_0 \left[\frac{1 + T_i(u_i + v_i)^{T_i+1} - (T_i + 1)(u_i + v_i)^{T_i}}{(1 - u_i - v_i)^2} - \right. \\ &\qquad \left. (u_i + v_i)^{T_i} \frac{(1 - T_i(1 - v_i))}{(1 - v_i)^2} \right] \end{aligned} \tag{13}$$

where

$$\begin{aligned} u_i &= (\phi_i + \gamma_i)/\mu_i, \\ v_i &= \gamma_i/\mu_i, \\ P_0 &= (1 - u_i - v_i)(1 - v_i)/[1 - v_i - (u_i + v_i)^{T_i}(u_i - v_i)], \\ T_i &= [\ln(\phi_i - \beta_i) + \ln(1 - v_i) - \ln(\phi_i(1 - v_i) - \beta_i(u_i - v_i))]/\ln u_i. \end{aligned}$$

We make the following observations regarding the behavior of $F_i^{(\ell)}(\beta_i, \gamma_i)$ and $F_i^{(r)}(\beta_i, \gamma_i)$ for both threshold policies.

1. $F_i^{(\ell)}(0,0) = F_i^{(r)}(0,0) = 1/\mu_i$,

2. $\lim_{\beta_i \uparrow \phi_i} \partial F^{(\ell)}(\beta_i, 0)/\partial \beta_i = \infty$.

This last observation is particularly interesting. Its consequence is that all nodes using either of the above two policies will always benefit from transferring a non-zero fraction of its jobs elsewhere to be processed. Both observations will be used in the coming analysis.

2.3. Optimal Solution

In this section we study the optimal solutions to problem (2) under the following assumptions

1. $F_i^{(\ell)}(\beta_i, \gamma_i)$ and $F_i^{(r)}(\beta_i, \gamma_i)$ are strictly increasing, differentiable, and convex functions of β_i and γ_i, $1 \le i \le N$,
2. $F_i^{(\ell)}(0,0) = F_i^{(r)}(0,0) = 1/\mu_i$,
3. $\lim_{\beta_i \uparrow \phi_i} \partial F^{(\ell)}(\beta_i, 0)/\partial \beta_i = \infty$.

We introduce two *incremental node delay* functions. These are

$$f_i(\beta_i, \gamma_i) = \beta_i \partial F_i^{(\ell)}(\beta_i, \gamma_i)/\partial \beta_i + \gamma_i \partial F_i^{(r)}(\beta_i, \gamma_i)/\partial \beta_i + F_i^{(\ell)}(\beta_i, \gamma_i), \qquad (14)$$

and

$$h_i(\beta_i, \gamma_i) = \beta_i \partial F_i^{(\ell)}(\beta_i, \gamma_i)/\partial \gamma_i + \gamma_i \partial F_i^{(r)}(\beta_i, \gamma_i)/\partial \gamma_i + F_i^{(r)}(\beta_i, \gamma_i). \qquad (15)$$

Here $f_i(\beta_i, \gamma_i)$ is the incremental change in delay due to an increase in local job flow at node i. Similarly, $h_i(\beta_i, \gamma_i)$ is the incremental change in delay due to an increase in remote job flow at node i. Note that assumption 2) implies that $f_i^{(\ell)}(0,0) = f_i^{(r)}(0,0)$ and that assumption 3) implies that $\lim_{\beta_i \uparrow \phi_i} f_i(\beta_i, \gamma_i) = \infty$, $0 \le \gamma_i \le \phi_i$. We now state the conditions that the solution to problem (2) must satisfy.

Theorem 1 *The optimal solution to problem (2) satisfies the relations*

$$\begin{aligned}
f_i(\beta_i, \gamma_i) &\ge \alpha + 1/\eta, & \beta_i;\ \gamma_i = 0, && (16a)\\
f_i(\beta_i, \gamma_i) &= \alpha + 1/\eta, & 0 < \beta_i < \phi_i,\ 0 \le \gamma_i, && (16b)\\
h_i(\beta_i, \gamma_i) &= \alpha, & 0 \le \beta_i < \phi_i;\ 0 < \gamma_i, && (16c)\\
h_i(\beta_i, \gamma_i) &\ge \alpha, & 0 \le \beta_i < \phi_i;\ \gamma_i = 0. && (16d)
\end{aligned}$$

subject to the total flow constraint

$$\sum_{i=1}^{N}(\beta_i + \gamma_i) = \Phi \tag{17}$$

where α is a Lagrangian multiplier.

The proof of this theorem is similar to that of Theorem 2 found in [TANT85] and is omitted. Conditions (16a) and (16b) describe the behavior of each node as a *source* of jobs to other nodes. One observes that in this sytem, *all* nodes behave as sources. This is due to the assumption that the incremental delay due to local jobs is unbounded in the range $(0, \phi_i)$. Condition (16a) is satisfied by a node that transfers *all* jobs to other nodes (idle source) whereas condition (16b) is satisfied by sources that process some local jobs (active source). Conditions (16c) and (16d) describe the behavior of a node as a recipient of remote jobs. Here a node will be a *sink*, i.e., will receive jobs, if conditon (16c) holds. Otherwise, if condition (16d) holds, it will not.

At this point in time we have not developed an algorithmic solution to problem (2) as we did for problem (4). We introduce one additional assumption in order to simplify our task of developing such an algorithm. We assume that the system consists of M clusters of nodes. Each cluster contains two or more identical nodes, i.e, identical arrival processes and throughput delay characteristics. Let N_i denote the number of nodes in the $i-th$ cluster and let the delay functions, incremental delay functions, etc. correspond to each node in cluster i.

Although the nodes do not separate into mutually exclusive sets of sources, sinks, and neutrals, the clusters do. Here a cluster is a source if it transfers some jobs to nodes in other clusters. A cluster is a sink if its constituent nodes process jobs that originated at other clusters. Last, a cluster is neutral if it neither sends jobs elsewhere nor processes jobs from other clusters. The set of source clusters further divide into two mutually exclusive sets of idle source clusters and active source clusters.

Theorem 2 *The clustering property is a sufficient condition for problem (2) to have an optimal solution where each cluster is either a source, a sink, or neutral.*

The proof is similar to the proof of Theorem 1 in [TANT85] and is omitted from this paper. Figure 3 illustrates the behavior of the system when it consists of clusters of identical nodes.

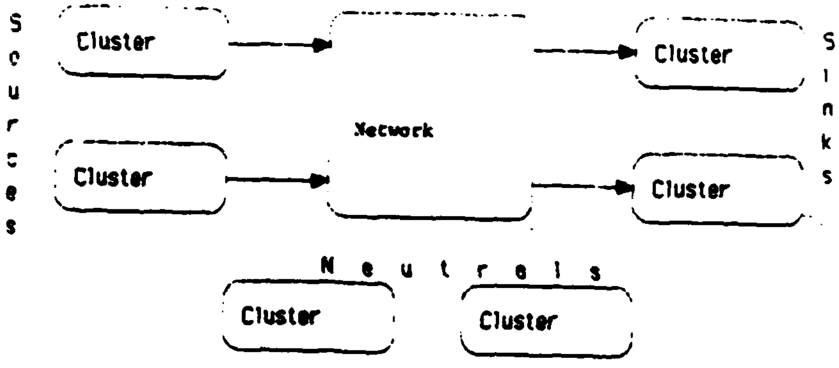


Figure 3: System partitioned into source, sink and neutral clusters.

The formulation of this problem can now be simplified and placed into a form so that the results described in Section 3 can be applied. Define θ_i to be the flow of completed jobs in the i^{th} cluster. Define $D(\beta_i,\theta_i)$ to be the average delay of jobs processed at the i^{th} cluster including their communication delays when the total job completion rate is θ_i and the local completion rate is β_i. This delay is expressed as

$$D_i(\beta_i,\theta_i) = \left(\beta_i F_i^{(\ell)}(\beta_i,\theta_i-\beta_i) + (\theta_i-\beta_i)\left[F_i^{(r)}(\beta_i,\theta_i-\beta_i)+1/\eta\right]\right)/\theta_i \qquad (18)$$

If we are given the value of θ_i for the optimal solution of problem (2), then we know that β_i must be the solution to

$$\begin{aligned} \text{minimize} \quad & D_i(\beta_i,\theta_i) \\ \text{subject to} \quad & 0 \leq \beta_i \leq \phi_i, \end{aligned} \qquad (19)$$

The optimal value of γ_i is $\gamma_i = \theta_i - \beta_i$.

Define $F_i(\theta_i)$ to be the the minimum job delay for a given θ_i, i.e., $F_i(\theta_i) = \min_{0\leq\beta_i<\phi_i} D_i(\beta_i,\theta_i)$. Since $F_i^{(\ell)}(\beta_i,\gamma_i)$ and $F_i^{(r)}(\beta_i,\gamma_i)$ are convex functions, it follows that $F_i(\theta_i)$ is convex. Let $\Theta = [\theta_i]$. We can now rewrite problem (2) as

$$\begin{aligned} \text{minimize} \quad & D(\Theta) = \textstyle\sum_{i=1}^{M} \theta_i F_i(\theta_i)/\Phi + \lambda/(\eta\Phi) \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{M} \theta_i = \Phi, \\ & 0 \leq \theta_i, \qquad\qquad 1 \leq i \leq M, \end{aligned} \qquad (20)$$

where λ is the inter-cluster job transfer flow which may be expressed as

$$\lambda = \frac{1}{2}\sum_{i=1}^{M} |N_i\phi_i - \theta_i|. \tag{21}$$

This problem formulation is identical to the formulation of problem (4). Consequently, the solution algorithms developed in [TANT85] can be applied to this problem. One of these results which we state in the context of our problem is

Theorem 3 *The intercluster traffic, λ, is a nonincreasing function of the network delay $1/\eta$.*

We make the following conjecture regarding the behavior of the *total* network traffic.

Conjecture 1 *The total network traffic, γ, is a nonincreasing function of the network delay $1/\eta$.*

Although we have been unable to prove this in general, it is easy to show when the system consists of a single cluster.

Theorem 4 *The total network traffic, γ, within a single cluster system ($M = 1$) is a nonincreasing function of the network delay $1/\eta$.*

Proof: The proof follows from the convexity of $F_i^{(\ell)}(\theta_i)$ and the fact that the network delay contribution is a linear function of γ.

3. LOAD BALANCING POLICIES USING GLOBAL STATE INFORMATION

In this section, we describe and analyze the behavior of a forward probing threshold lb policy where we account for communication delays. The forward probing (FP) lb policy operates in the following manner.

Forward probing: Under this policy a node processes a job locally if the total number of jobs in the queue is less than or equal to $T + 1$ where T is a threshold parameter of the policy. If the number of jobs exceeds $T + 1$, then an attempt is made to transfer the newly arrived job to another node. A finite number, L_p, of nodes (usually $L_p = 2$ or 3 is adequate) is probed at random to determine a placement for the job. A probed node responds positively if the number of jobs it possesses is less than $T + 1$ and it is not already waiting for some other remote job. If more than one node responds positively, the sender node transfers the job to one of these respondents, picked at random. If none of the probed nodes responds positively, i.e., this probe was unsuccessful, the node waits for another local arrival before it can probe again.

3.1. Mathematical Analysis

It is assumed that the job arrival process at each node is Poisson, with parameter ϕ. Also, the service times and job transfer times are assumed to be exponentially distributed, with means $1/\mu$ and $1/\eta$, respectively. The job transfer time includes the time between the initiation of a transfer from a node and the successful reception of the job at the destination node. The time to transfer a probe and obtain its response is assumed to be negligible. The impact of this assumption is discussed in [MIRC87b].

The nodes are assumed to be homogeneous and jobs are assumed to be executed on a First-Come-First-Served (FCFS) basis at each node.

An exact analysis of this system is not feasible. Consequently, we decompose the model such that the model for each node can be solved independently of the others [EAGE86]. The interactions between the nodes which result in job transfers for the purpose of load sharing in the distributed system, are modelled by means of modifications to the arrival and/or departure process at each node. These interactions will be described in detail further in this subsection.

We conjecture that the method of decomposition is asymptotically exact as the number of nodes tends to infinity. Actual experimental results indicate that there exists very good agreement between the model and simulations even when the systems are of relatively small size (= 10 nodes). Thus, the approximation is likely to be even better for larger systems.

The state of the node is represented by a tuple (N_t, J_t), where N_t is the number of jobs at a node and J_t indicates if the node is being probed or not. Here J_t takes on value 0 if the node is not being probed and value 1 if it is being probed. We are interested in $N = \lim_{t\to\infty} N_t$ and $J = \lim_{t\to\infty} J_t$ when the limits exist.

The analysis of the algorithms is performed using the Matrix-geometric solution technique [NEUT81], which yields an exact solution of the model for each node. The material in this paper involves a Jacobi matrix, whose detailed definition will be provided as in Latouche [LATO81]. A matrix such as

$$\left[\begin{array}{cccccccc} b_0 & c_0 & 0 & 0 & & & \cdots & \\ a_1 & b_1 & c_1 & 0 & & & \cdots & \\ 0 & a_2 & b_2 & c_2 & & & & \\ \cdots & \cdots & & & a_{m-2} & b_{m-2} & c_{m-2} & 0 \\ \cdots & \cdots & & & 0 & a_{m-1} & b_{m-1} & c_{m-1} \\ \cdots & \cdots & & & 0 & 0 & a_m & b_m \end{array}\right]$$

will be displayed as

$$\left\|\begin{array}{ccccccc} & c_0 & c_1 & \cdots & c_{m-3} & c_{m-2} & c_{m-1} \\ b_0 & b_1 & b_2 & \cdots & b_{m-2} & b_{m-1} & b_m \\ a_1 & a_2 & a_3 & \cdots & a_{m-1} & a_m & \end{array}\right\|$$

We define

$$\begin{aligned} y(n,j) &= \lim_{t\to\infty} P(N_t = n,\ J_t = j),\ 0 \le n,\ 0 \le j \le 1, \\ \mathbf{p_n} &= (y(n,0), y(n,1)),\ 0 \le n, \\ \vec{\mathbf{p}} &= (\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, ...\mathbf{p_i}, ...). \end{aligned}$$

If the Markov process (N_t, J_t) is ergodic then $\vec{\mathbf{p}}$ is its steady state probability vector satisfying $\vec{\mathbf{p}}Q = 0$, where Q is the infinitesimal generator this Markov process. Q, the infinitesimal generator for the FP algorithm, has the structure of a block-tridiagonal matrix of the form

$$Q = \left\|\begin{array}{cccccccc} & B_{01} & \cdots & B_{01} & B_{01} & A_0 & A_0 & \cdots \\ B_{00} & B_{11} & \cdots & B_{11} & A_1 & A_1 & A_1 & \cdots \\ A_2 & A_2 & \cdots & A_2 & A_2 & A_2 & A_2 & \cdots \end{array}\right\|$$

with exactly $T-1$ columns of (B_{01}, B_{11}, A_2). We define the matrices $B_{00}, B_{01}, B_{11}, A_2$, A_1 and A_0 in Appendix A.

In the subsequent discusssion, h is the probability of failure in finding an assignment for a spare job in response to a set of forward probes. Thus, $\bar{h} = 1 - h$, is the probability that at least one of the probed nodes will accept a remote job. The rate at which a node receives forward probes is denoted by δ.

Neuts[NEUT81] examined Markov processes with generators such as Q and determined the conditions for ergodicity. The general conditions for stability were derived for those cases where the infinitesimal generator $A = A_0 + A_1 + A_2$, corresponding to the geometric part of the Markov Process is irreducible. Since this is not the case here (i.e., A is reducible), one has to explicitly determine the stability criterion for the process. From Neuts [NEUT81], one is able to show that the queue is ***stable*** if and only if $\phi h < \mu$.

We now assume that all the values of all the parameters (i.e., h and δ) are known. We know from Neuts[NEUT81] that

$$\mathbf{p}_i = \mathbf{p}_{T+1} R^{T+1-i}, \forall i \geq T+1$$

Thus,

$$\sum_{i \geq T+1} \mathbf{p}_i = \mathbf{p}_{T+1}(I - R)^{-1}$$

Also,

$$(\sum_{i=0}^{T} \mathbf{p}_i + \mathbf{p}_{T+1}(I - R)^{-1})e = 1$$

where R is the minimal solution of

$$A_0 + RA_1 + R^2 A_2 = 0$$

with $R \geq 0$ and the spectral radius of $R, sp(R) < 1$, and I is the identity matrix. The following iterative procedure is used to compute R [NELS85]:

$$R(0) = 0$$

$$R(n+1) = -A_0 A_1^{-1} - R(n)^2 A_2 A_1^{-1}, n \geq 0.$$

The probabilities $\mathbf{p}_1, \cdots, \mathbf{p}_{T+1}$ are obtained by solving the following system of linear equations,

$$(\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{T+1}) \left\| \begin{array}{ccccc} & B_{01} & ... & B_{01} & B_{01} \\ B_{00} & B_{11} & ... & B_{11} & A_1 + RA_2 \\ A_2 & A_2 & ... & A_2 & \end{array} \right\| = 0$$

where the number of columns in the matrix is exactly $T + 1$.

The expected number of jobs at a node, $E[N]$, and the expected response time of a job, $E[D]$, are given by the following expressions:

$$\begin{aligned} E[N] &= \sum_{i\geq 1} i\,\mathbf{p}_i\,e \\ &= (\mathbf{p}_{T+1}(I-R)^{-2} + T\mathbf{p}_{T+1}(I-R)^{-1})e \\ E[D] &= \frac{(E[N] + \frac{\gamma}{\eta})}{\phi} \end{aligned}$$

where γ is the flow of remote jobs into a node.

We are left with determining the unknown parameters, h and δ. Here h can be represented as $h = x^{L_p}$ where L_p is the number of nodes probed and x is the probability that a particular node will respond negatively to a probe. This is given as

$$x = \sum_{i\leq T} \mathbf{p}_i\,[01]^T + \sum_{i>T} \mathbf{p}_i e.$$

The value of the parameter δ is obtained by equating the flow of remote jobs out of a node to the flow of remote jobs in to a node. If we denote these two flows as $FFRO$ and $FFRI$ respectively where

$$\begin{aligned} FFRO &= \phi\bar{h}\sum_{i>T} \mathbf{p}_i\,[11]^T, \\ FFRI &= \delta\sum_{i\leq T} \mathbf{p}_i\,[10]^T, \end{aligned}$$

then δ is given as

$$\delta = \frac{FFRO}{\sum_{i\leq T}\mathbf{p}_i\,[10]^T}.$$

We have used an iterative algorithm based on solving the above equations in order to obtain the values of h and δ. The reader is referred to [MIRC87a] for further details.

3.2 Numerical Results

Figure 4 shows the behavior of the FP policy as the communication delay, $1/\eta$ increases for different system loads. Each point was obtained for the optimal threshold value of T. We have also plotted the mean response time for a threshold policy that

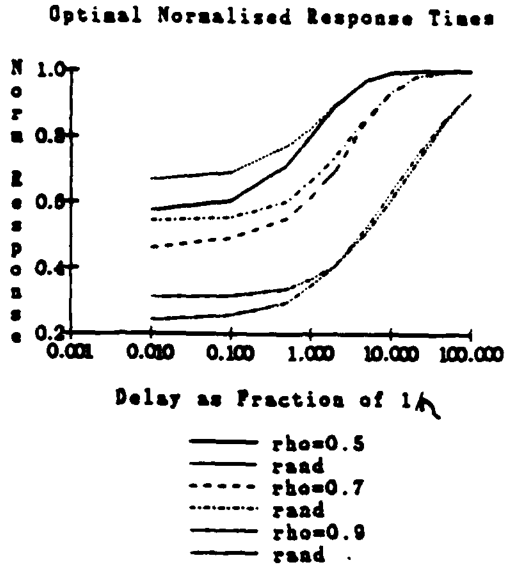


Figure 4: Response Time as a Function of Communication Delay.

chooses destinations randomly. FP provides significantly lower response times for different loads when the communication delay is low. It is interesting to observe that if system loads are high, FP still provides an improvement in performance when the communication delay is high, ($10/\mu \leq 1/\eta \leq 100/\mu$). The state information acquired with probes provides improved performance over the random policy when communication delays are low ($1/\eta \leq 3/\mu$). The difference in performance disappears once the communication delay falls outside this range. Consequently, global state information ceases to be useful in these cases.

Figure 5 shows the effect of communications delay on the network traffic (jobs and probes) generated by a node operating at its optimal threshold. It can be seen that the network traffic quickly tends to zero for low to moderate loads for $1/\eta \leq 2/\mu$. The effect is similar for traffic intensity of 0.9, but the decrease occurs more slowly. Thus, very little or almost no load sharing occurs at high delays.

4. SUMMARY

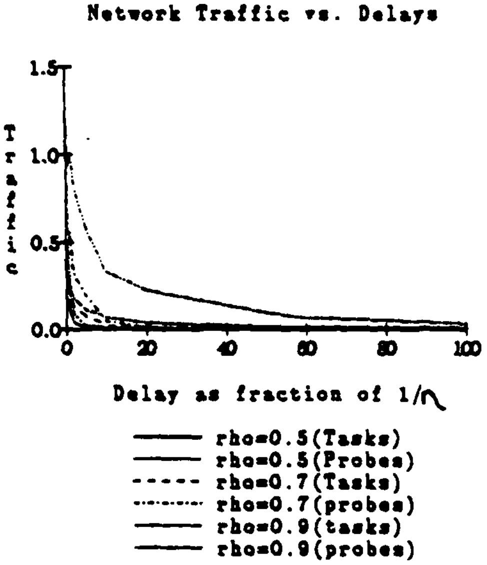


Figure 5: Network Traffic as a Function of Communication Delay.

We have shown in Section 2. the effects of increasing communication delay on the behavior of a wide class of lb policies. However, this is based on the assumption that the local job flow can take any value. We believe that real lb policies that use integer valued thresholds will show a similar type of behavior. Further study is required to substantiate this. We have also shown that the simple algorithms developed earlier in the context of static probabilistic load balancing can be applied to systems using more complicated lb policies provided that they divide into clusters containing two or more identical nodes. This should be useful as there exist many examples of such systems.

In Section 3. we studied the effects of global state information on the performance of lb policies. Our results indicate that global state information does not improve performance once the communication delay is larger than three times the processing time. This result holds, however, for homogeneous systems executing homogeneous workloads. It remains for a future study to focus on heterogeneous systems.

Appendix A In this appendix, we provide closed form representations for the matrices in the case of the Forward probing algorithm.

$$B_{00} = \begin{bmatrix} -(\delta+\phi) & \delta \\ 0 & -(\eta+\phi) \end{bmatrix}, \quad B_{01} = \begin{bmatrix} \phi & 0 \\ \eta & \phi \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} -(\delta+\phi+\mu) & \delta \\ 0 & -(\mu+\eta+\phi) \end{bmatrix}, \quad A_0 = \begin{bmatrix} \phi h & 0 \\ \eta & \phi h \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -(\mu+\phi h) & 0 \\ 0 & -(\mu+\phi h+\eta) \end{bmatrix}, \quad A_2 = \mu I_2$$

where I_2 is the identity matrix of size 2.

References

[EAGE86] Eager, D., E. Lazowska, and J.Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Soft. Engg.*, Vol. SE-12, No. 5, pp. 662–675, May 1986.

[LATO81] Latouche, G., "Algorithmic Analysis of a Multiprogramming-Multiprocessor Computer System," *J. ACM*, Vol. 28, October 1981.

[LEE87] Lee, K. J., *Load Balancing in Distributed Computer Systems*, PhD thesis, ECE Dept., University of Massachusetts, February 1987.

[LIVN82] Livny, M. and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Performance Evaluation Review*, Vol. 11, No. 1, pp. 47–55, 1982.

[MIRC87a] Mirchandaney, R., D. Towsley, and J. A. Stankovic, "Analysis of the Effects of Delays on Load Sharing," *IEEE Trans. Computers*, 1987. submitted for review.

[MIRC87b] Mirchandaney, R., D. Towsley, and J. A. Stankovic, "Effects of Delays on Receiver-Initiated Load Sharing Policies," *Journal of Parallel and Distributed Computing*, 1987. submitted for review.

[NELS85] Nelson, R. and B. R. Iyer, "Analysis of a Replicated Data Base," *Performance Evaluation*, Vol. 5, pp. 133–48, 1985.

[NEUT81] Neuts, M. F., *Matrix-Geometric solutions in Stochastic Models: An Algorithmic Approach, Mathematical Sciences*, Johns Hopkins University Press, 1981.

[TANT85] Tantawi, A. and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *J. ACM*, Vol. 32, pp. 445–465, Apr. 1985.

[WANG85] Wang, Y. and R. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Computers*, Vol. C-34, March 1985.

A Comparison of the Processor Sharing and First Come First Serve Policies for Scheduling Fork-Join Jobs in Multiprocessors*

D. Towsley
Dept. Computer and Information Sciences
Univ. of Massachusetts
Amherst, MA 01003

C. G. Rommel
Department of Electrical Engineering
Univ. of Massachusetts
Amherst, MA 01003

J. A. Stankovic
Dept. Computer and Information Sciences
Univ. of Massachusetts
Amherst, MA 01003

July 28, 1987

Abstract

In this paper a model of a shared memory multiprocessor that executes *fork-join* parallel programs as a bulk arrival $M^X/M/c$ queueing system is developed. Here a fork-join job is one that consists of a set of X tasks. All of the tasks arrive simultaneously to the system and the job is assumed to complete when the last task completes. We develop tight upper and lower bounds for the mean response time of such programs when the scheduling discipline is processor sharing under the assumptions of exponential task service times and a Poisson job arrival process. We study two processor sharing policies, one called *task scheduling* processor sharing and the other called *job scheduling* processor sharing. The first policy schedules tasks independently of each other and allows parallel execution, whereas the second policy schedules entire jobs as a unit and thereby does not allow parallel execution of an individual program. We find that the

*This work was supported in part by the National Science Foundation under grant MCS-8104203 and by RADC under contract RI-44896X.

job scheduling policy exhibits better performance than task scheduling only on systems with a small number of processors, where the system is operating at high loads and is executing programs that can sustain a large degree of parallelism. Consequently, in general, task scheduling outperforms job scheduling. We also compare the performance of the processor sharing policy with first come first serve. We find that first come first serve exhibits better performance over a wide range of systems. The paper also studies the performance of processor sharing and first come first serve with two classes of jobs, and when a specific number of processors is statically assigned to each of these classes.

1 Introduction

With the advent of multiprocessors[Ost86] and programming languages that support parallel programming, (e.g., Concurrent Pascal [Han75], CSP[Hoa85], and Ada [Pyl81]) there is increasing interest in modeling the performance of parallel programs. In this paper, we evaluate the performance of a particular type of parallel program, a *fork-join* job, on a multiprocessor consisting of identical processors when the service discipline is processor sharing. In our model a fork-join job is composed of a set of tasks each of which can be scheduled independently of the others at any processor. All tasks in a given job arrive simultaneously to the system. The job completes when the last task completes.

The performance of parallel programs such as fork-join jobs is significantly affected by the choice of policy that is used to schedule tasks. We analyze the performance of a processor sharing (PS) policy that schedules *tasks* of a job independently of each other. We refer to this policy as *task scheduling PS*, TS-PS. We compare the performance of this TS-PS policy to that of a second PS policy that schedules enitre *jobs* (as a single unit) independently of each other. We refer to this policy as *job scheduling PS*, JS-PS. The TS-PS policy is unaware that jobs exist whereas the JS-PS policy is unaware that tasks exist. We also compare the performance of TS-PS and JS-PS to the first come first serve (FCFS) policy. In these comparisons we consider different numbers of processors, sizes of fork-join jobs, multiple classes, and dedicated assignments of the processors of the multiprocessor to the different classes.

In the course of our study, we develop upper and lower bounds on the mean fork-join job response times under TS-PS. These bounds are generally very tight and we approximate the mean job response time by taking the average of the two bounds. Analyses of the other two policies, JS-PS and FCFS have already appeared in the literature ([RTS87,NTT87]).

We make the following observations from our study.

- FCFS provides better performance than TS-PS or JS-PS for a wide range of workloads and number of processors. It appears that the advantages that FCFS has over PS in single processor systems carries over to multiprocessors executing parallel programs. This carries the implication that one should choose large quantum sizes for round robin policies operating on multiprocessors.

- TS-PS performs better than JS-PS most of the time. However, if the number of processors is small, the degree of parallelism high, and the processor utilization is high, JS-PS can perform better. This same phenomenon was observed on single processors in an earlier study, [RTS87].
- It may be useful to partition the processors in a multiprocessor into separate pools to handle different classes of jobs rather than having the jobs share the processors. We observe that jobs requiring the least amount of computation can benefit from such a partition.

In the remainder of this section we briefly review earlier work and outline the remainder of this paper. Processor-sharing has been addressed in the literature in several ways since its introduction [Kle64]. A survey of processor-sharing results may be found in [Kle76]. An exact analysis of the TS-PS policy operating on a *single* processor was performed by Rommel, et al. [RTS87]. Unfortunately, the approach used in that paper does not extend to multiple processors. This study first demonstrated that job scheduling can give better performance than task scheduling. In addition, there is a growing literature on fork-join queueing systems [BM85,BMT87,NT85]. Although these referenced papers analyze fork-join jobs, their analysis differs from that studied in this paper in that processors are allocated to specific tasks prior to execution. We are interested in systems where processors can be *dynamically* allocated to different tasks.

The format of this paper is as follows. We describe the queueing system under consideration in Section 2. Section 3 contains expressions for the upper and lower bounds on the mean response time for the TS-PS scheduling policy along with an approximate analysis of that policy. This is followed by our numerical results in Section 4. Finally, in Section 5 we summarize the results of the paper.

2 Model Description

We consider a system of c identical processors that serve a single queue. Fork-join jobs enter the system according to a Poisson process with parameter λ. A fork-join job consists of X tasks that can be processed independently of each other where X is a random variable (r.v.) with probability distribution $\alpha_i = P[X = i]$, $i = 1, 2, \cdots$. The service time required by a task is assumed to be an exponential r.v. with parameter μ and is independent of the service requirements of all other tasks.

We are interested in the steady state behavior of this system when operating under the task scheduling processor sharing (TS-PS) and the job scheduling processor sharing (JS-PS) policies. As described in section 1, TS-PS is a policy that performs processor sharing at the task level and JS-PS is a policy that performs processor sharing at the job level. Thus, if the system contains two jobs, one with one task, the other with three tasks, then TS-PS provides an equal amount of service to each task and is capable of utilizing four processors. In this same example JS-PS sees two jobs, one whose service time is that of a single task, the other whose service time is the sum of the service times of the three tasks. JS-PS provides equal service to the two jobs and is only able to utilize two processors.

In both cases, we focus on the response time of a random job, i.e., the interval of time measured from the arrival of a job until the service completion of the *last task* associated with that job. The system can be visualized as a queue for tasks, c servers, and a waiting area for tasks that have completed service but are awaiting the completion of the last task associated with the job (Figure 1). This last queue is sometimes referred to as the *synchronization queue*. We denote this response time as T.

3 Analysis

In this section we concern ourselves with obtaining the mean response time $E[T]$ under both TS-PS and JS-PS. We consider JS-PS first as it is the simplest to analyze.

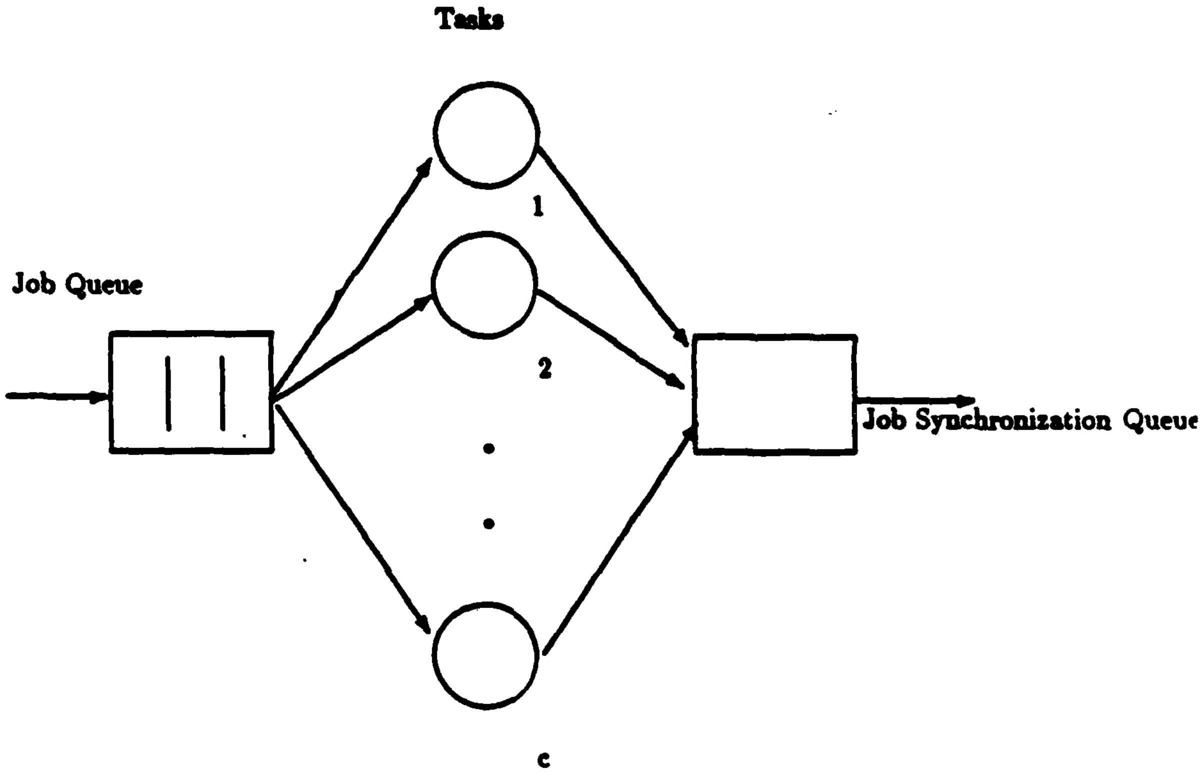


Figure 1: System Model

3.1 The JS-PS policy

Let L denote the number of jobs in the system under JS-PS. The distribution of L is identical to the queue length distribution of an $M/M/c$ system with arrival rate λ and average service time $E[X]/\mu$. Consequently, the average response time, $E[T]$, is ([All78])

$$E[T] = \frac{u^c(E[X]/\mu)}{cc![u^c/c! + (1-u/c)\sum_{n=0}^{c-1} u^n/n!](1-u/c)} + E[X]/\mu. \qquad (1)$$

where $u = \lambda E[X]/\mu$. $E[T] = E[L]/\lambda$.

3.2 The TS-PS policy

To analyze the TS-PS policy, consider the delay that a randomly selected job incurs. Let J denote this job. Let N be a r.v. that denotes the number of tasks in the system at the time that J arrives. Let $\pi_n = P[N = n]$, $n = 0, 1, \cdots$ denote the stationary distribution of N. Let $t_{i,n}$ denote the mean response time of J conditioned on the event that J consists of i tasks and that the system contains $N = n$ tasks at the time of its arrival, i.e. $t_{i,n} = E[T|X = i, N = n]$. We can write the following expression for the mean job response time,

$$E[T|X = i] = \sum_{n=0}^{\infty} \pi_n t_{i,n}, \quad i = 1, \cdots \qquad (2)$$

Removal of the number of tasks in J yields

$$E[T] = \sum_{i=1}^{\infty} \alpha_i E[T|X = i]. \qquad (3)$$

As described above, the number of tasks in the system is described by a Markov process. Fortunately, the behavior of this Markov process is independent of the policy used to schedule tasks so long as the policy does not schedule jobs based on service time information. Consequently, the distribution of N is identical to that for a bulk arrival $M^X/M/c$ system that schedules tasks in a FCFS manner. Expressions for the queue length distribution for this system can be found in earlier papers [CT83,Yao85,NTT87] and are omitted here.

We focus on the conditional expectations $t_{i,n}$. We define a Markov Chain with state (I_t, M_t) with infinitesimal generator Q where I_t is the number of tasks remaining in J at time t after J is introduced at time 0, M_t is the number of tasks in the system at time t that are not part of J, and $\mathbf{Q} = [q_{(i,n),(l,m)}]$ where

$$q_{(i,n),(l,m)} = \begin{cases} \frac{i}{i+n}\mu_{i+n}, & l = i-1,\ n = m, \\ \frac{n}{i+n}\mu_{i+n}, & l = i,\ m = n-1, \\ \lambda\alpha_{m-n}, & i = l,\ m > n, \\ -(\lambda + \mu_{i+n}), & i = l,\ m = n, \\ 0 & ,otherwise \end{cases} \tag{4}$$

where

$$\mu_k = \begin{cases} k\mu, & k = 1, \cdots, c \\ c\mu, & k = c+1, \cdots. \end{cases}$$

The resulting chain is transient. Figure 2 illustrates the associated state diagram when all jobs consist of exactly 2 tasks. If we define $T_{(i,n),(l,m)}$ to be the time that elapses between entering state (i,n) and entering state (l,m), then the conditional response time, $t_{i,n}$ can be expressed as

$$t_{i,n} = \sum_{m=0}^{\infty} E[T_{(i,n),(0,m)}], \quad i = 1, \cdots; \; n = 0, \cdots. \tag{5}$$

It follows from the definition of Q that $t_{i,n}$ satisfies

$$\begin{aligned} t_{1,0} &= \frac{1}{\lambda + \mu_1} + \frac{\lambda}{\lambda + \mu_1}\sum_{k=1}^{\infty}\alpha_k t_{1,k}, \\ t_{1,n} &= \frac{1}{\lambda + \mu_{n+1}} + \frac{\lambda}{\lambda + \mu_{n+1}}\sum_{k=1}^{\infty}\alpha_k t_{1,n+k} + \frac{n\mu_{n+1}/(n+1)}{\lambda + \mu_{n+1}} t_{1,n-1}, \quad n = 1, \cdots, \\ t_{i,0} &= \frac{1}{\lambda + \mu_i} + \frac{\lambda}{\lambda + \mu_i}\sum_{k=1}^{\infty}\alpha_k t_{i,k} + \frac{\mu_i}{\lambda + \mu_i} t_{i-1,0}, \quad i = 2, \cdots, \\ t_{i,n} &= \frac{1}{\lambda + \mu_{i+n}} + \frac{\lambda}{\lambda + \mu_{i+n}}\sum_{k=1}^{\infty}\alpha_k t_{i,n+k} \\ &\quad \frac{n\mu_{i+n}/(i+n)}{\lambda + \mu_{i+n}} t_{i,n-1} + \frac{i\mu_{i+n}/(i+n)}{\lambda + \mu_{i+n}} t_{i-1,n}, 1 = 2, \cdots; n = 1, \cdots. \end{aligned} \tag{6}$$

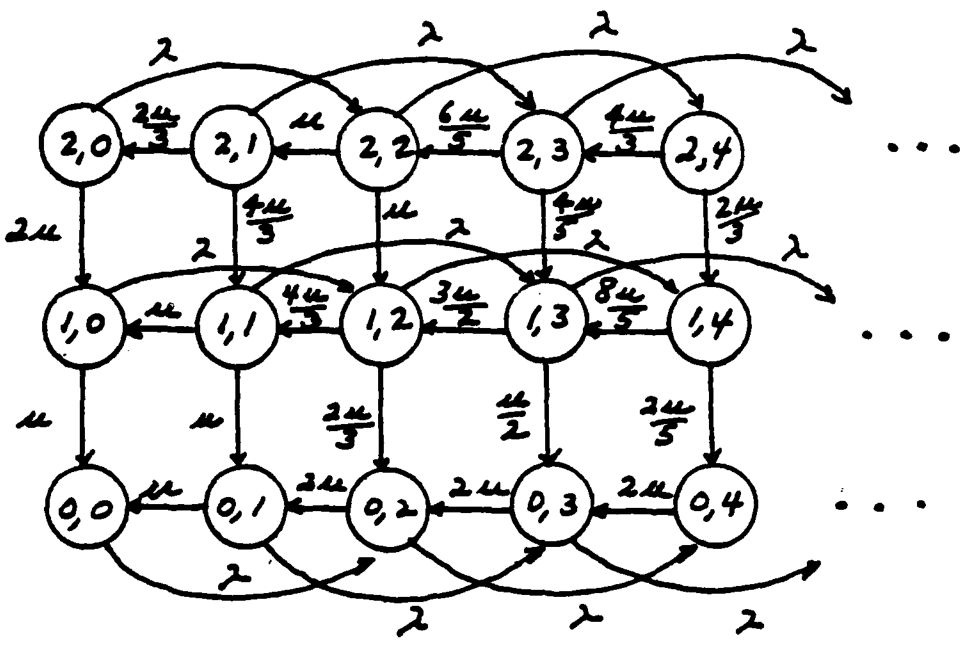


Figure 2: State diagram for the exact system when jobs consist of 2 tasks and $c = 2$.

Consider the last expression, $t_{i,n}$. The first term is the average time that the system spends in state (i, n). The second term is the contribution to $t_{i,n}$ due to an arrival. The third and fourth terms are the contributions due to a departure of a task belonging to J and a task not belonging to J, respectively.

We are unable to obtain a closed form solution to equation (6). As there are a countably infinite number of unknown variables $t_{i,n}$, $i = 1, \cdots; n = 0, \cdots$, it is impossible to obtain exact numerical values for these quantities. Consequently, the remainder of this section is concerned with developing upper and lower bounds on the conditional expectations $t_{i,n}$. These can be used to obtain upper and lower bounds for $E[T|X = i]$, $i = 1, \cdots$. We treat each in turn.

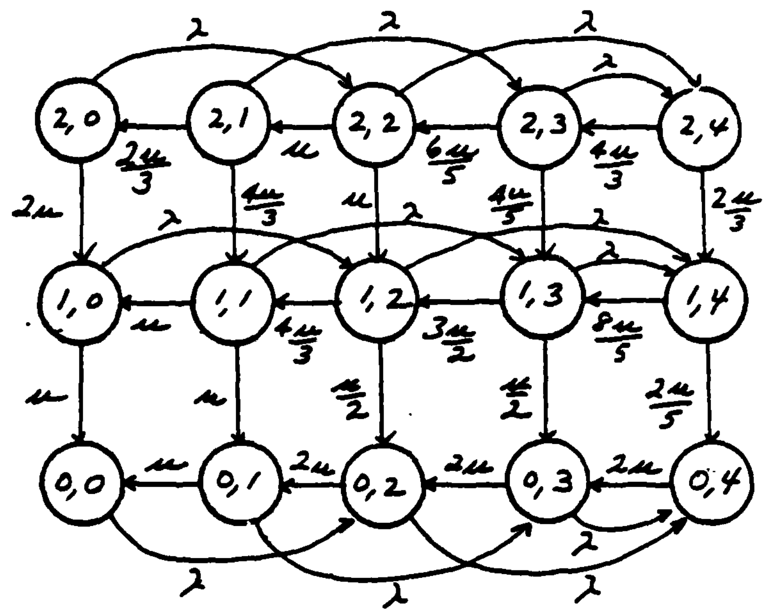


Figure 3: The state diagram associated with the lower bound, 2 tasks per job.

3.2.1 A Lower bound on $E[T|X = i]$

We study a Markov chain with state $(I_t^{(lb)}, M_t^{(lb)})$ that yields lower bounds $t_{i,n}^{(lb)}$ on $t_{i,n}$, $i = 1, \cdots, n = 0, \cdots$. This chain has infinitesimal generator $\mathbf{Q}^{(lb)} = [q^{(lb)}_{(i,n),(l,m)}]$ where

$$q^{(lb)}_{(i,n),(l,m)} = \begin{cases} \frac{i}{i+n}\mu_{i+n}, & l = i-1,\ n = m;\ 0 \leq m \leq B, \\ \frac{n}{i+n}\mu_{i+n}, & l = i,\ m = n-1;\ 1 \leq m < B, \\ \lambda\alpha_{m-n}, & i = l,\ 0 \leq n < m < B, \\ \lambda\sum_{k=B-n}^{\infty}\alpha_k, & l = i,\ m = B, 0 \leq n < B, \\ -(\lambda + \mu_{i+n}), & i = l,\ m = n,\ 0 \leq m < B, \\ 0, & \textit{otherwise}. \end{cases} \qquad (7)$$

This Markov chain corresponds to a system in which no more than B tasks not belonging to J are allowed in. Consequently, this modified system has fewer tasks that do not belong to J than the original system. The response time of J will be less in this system. Figure 3 illustrates the state diagram for this Markov chain when each job consists of exactly 2 tasks.

The conditional expectations, $t_{i,n}^{(lb)}$ satisfy

$$
\begin{aligned}
t_{1,0}^{(lb)} &= \frac{1}{\lambda+\mu_1} + \frac{\lambda}{\lambda+\mu_1}\left(\sum_{k=1}^{B}\alpha_k t_{1,k}^{(lb)} + \sum_{k=B+1}^{\infty}\alpha_k t_{1,B}^{(lb)}\right), \\
t_{1,n}^{(lb)} &= \frac{1}{\lambda+\mu_{n+1}} + \frac{n\mu_{n+1}/(n+1)}{\lambda+\mu_{n+1}} t_{1,n-1}^{(lb)} + \\
&\quad \frac{\lambda}{\lambda+\mu_{n+1}}\left(\sum_{k=1}^{B-n}\alpha_k t_{1,n+k}^{(lb)} + \sum_{k=B-n+1}^{\infty}\alpha_k t_{1,B}^{(lb)}\right), \quad n = 1,\cdots,B, \\
t_{i,0}^{(lb)} &= \frac{1}{\lambda+\mu_i} + \frac{\mu_i}{\lambda+\mu_i} t_{i-1,0}^{(lb)} + \\
&\quad \frac{\lambda}{\lambda+\mu_i}\left(\sum_{k=1}^{\infty}\alpha_k t_{i,k}^{(lb)} + \sum_{k=B+1}^{\infty}\alpha_k t_{i,B}^{(lb)}\right), \quad i = 2,\cdots, \\
t_{i,n}^{(lb)} &= \frac{1}{\lambda+\mu_{i+n}} + \frac{\lambda}{\lambda+\mu_{i+n}}\left(\sum_{k=1}^{B-n}\alpha_k t_{i,n+k}^{(lb)} + \sum_{k=B-n+1}^{\infty}\alpha_k t_{i,B}^{(lb)}\right) + \\
&\quad \frac{n\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}} t_{i,n-1}^{(lb)} + \frac{i\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}} t_{i-1,n}^{(lb)}, 1 = 2,\cdots; n = 1,\cdots,B. \qquad (8)
\end{aligned}
$$

Last, $t_{i,n}$, $n > B$ is bounded from below by $t_{i,B}^{(lb)}$, i.e.,

$$t_{i,n}^{(lb)} = t_{i,B}^{(lb)}, \quad i = 1,\cdots; n = B+1,\cdots. \qquad (9)$$

Thus we have the following lower bound on $E[T|X = i]$,

$$E[T|X = i] \leq \sum_{n=0}^{B-1} \pi_n t_{i,n}^{(lb)} + P[N \geq B] t_{i,B}^{(lb)}, \quad i = 1,\cdots. \qquad (10)$$

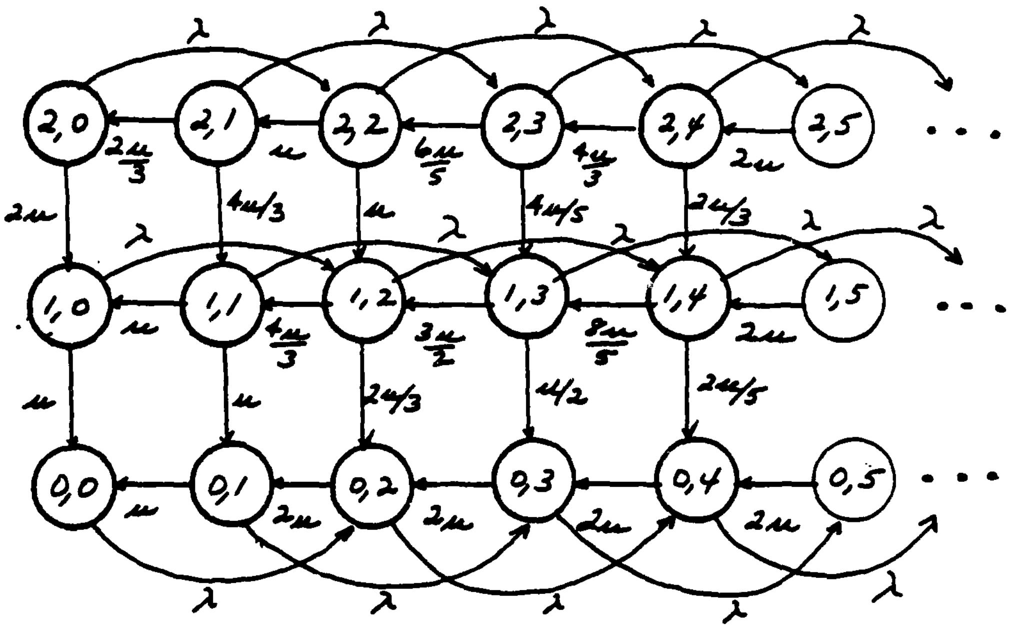


Figure 4: State diagram for upper bound, $B = 4$.

3.2.2 An upper bound on $E[T|X = i]$

We study a Markov chain with state $(I_t^{(ub)}, M_t^{(ub)})$ that yields upper bounds $t_{i,n}^{(ub)}$ on $t_{i,n}$, $i = 1, \cdots, n = 0, \cdots$. This chain has infinitesimal generator $\mathbf{Q}^{(ub)} = [q_{(i,n),(l,m)}^{(ub)}]$ where

$$q_{(i,n),(l,m)}^{(ub)} = \begin{cases} \frac{i}{i+n}\mu_{i+n}, & l = i-1,\ n = m;\ 0 \le m \le B, \\ \frac{n}{i+n}\mu_{i+n}, & l = i,\ m = n-1;\ 1 \le m < B, \\ \mu_{i+n}, & l = i,\ m = n-1,\ B \le m, \\ \lambda\alpha_{m-n}, & i = l,\ 0 \le n < m < B, \\ \lambda\sum_{k=B-n}^{\infty}\alpha_k, & l = i,\ m = B, 0 \le n < B, \\ -(\lambda + \mu_{i+n}), & i = l,\ m = n,\ 0 \le m < B, \\ 0, & \textit{otherwise}. \end{cases} \tag{11}$$

This system behaves like the original system except when the number of tasks n not belonging to J exceeds B. In this case, the system *is not allowed to serve* J, but instead only serves the other tasks. This continues until the number of additional tasks falls to B at which point the system behaves like the original system. Figure 4 illustrates the behavior of this system when jobs contain exactly two tasks and $B = 4$.

Assume that $B \ge c$. Consider the situation where J contains i tasks and there are an additional $j < B$ tasks in the system. Now assume that k tasks arrive and that $n + k > B$. In this case, the time during which there are $B + 1$ or more additional tasks in the modified system is equal to the length of the busy period associated with a bulk arrival $M^X/M/1$ queue with rate μc that is initiated by the arrival of $n + k - B$ tasks. Consequently, we can write the following set of equations describing the expected response time of a job conditioned on the number of tasks at the time of arrival and the number of tasks in the arriving job, $t_{i,n}^{(ub)}$,

$$\begin{aligned}
t_{1,0}^{(ub)} &= \frac{1}{\lambda + \mu_1} + \frac{\lambda}{\lambda + \mu_1}\left(\sum_{k=1}^{B} \alpha_k t_{1,k}^{(ub)} + \sum_{k=B+1}^{\infty} \alpha_k (t_{1,B}^{(ub)} + b_{k-B})\right), \\
t_{1,n}^{(ub)} &= \frac{1}{\lambda + \mu_{n+1}} + \frac{\lambda}{\lambda + \mu_{n+1}}\left(\sum_{k=1}^{B-n} \alpha_k t_{1,n+k}^{(ub)} + \sum_{k=B-n+1}^{\infty} \alpha_k (t_{1,B}^{(ub)} + b_{n+k-B})\right) + \\
&\quad \frac{n\mu_{n+1}/(n+1)}{\lambda + \mu_{n+1}} t_{1,n-1}^{(ub)}, \quad n = 1, \cdots, B, \\
t_{i,0}^{(ub)} &= \frac{1}{\lambda + \mu_i} + \frac{\lambda}{\lambda + \mu_i}\left(\sum_{k=1}^{\infty} \alpha_k t_{i,k}^{(ub)} + \sum_{k=B+1}^{\infty} \alpha_k (t_{i,B}^{(ub)} + b_{k-B})\right) + \\
&\quad \frac{\mu_i}{\lambda + \mu_i} t_{i-1,0}^{(ub)}, \quad i = 2, \cdots, \\
t_{i,n}^{(ub)} &= \frac{1}{\lambda + \mu_{i+n}} + \frac{\lambda}{\lambda + \mu_{i+n}}\left(\sum_{k=1}^{B-n} \alpha_k t_{i,n+k}^{(ub)} + \sum_{k=B-n+1}^{\infty} \alpha_k (t_{i,B}^{(ub)} + b_{n+k-B})\right) + \\
&\quad \frac{n\mu_{i+n}/(i+n)}{\lambda + \mu_{i+n}} t_{i,n-1}^{(ub)} + \frac{i\mu_{i+n}/(i+n)}{\lambda + \mu_{i+n}} t_{i-1,n}^{(ub)}, 1 = 2, \cdots; n = 1, \cdots, B. \qquad (12)
\end{aligned}$$

where b_l is the average length of a busy period of an $M^X/M/1$ queue with arrival rate λ and service rate μc that is started by the arrival of i tasks. The value of b_l, $l = 1, \cdots$ is ([GH76])

$$b_l = \frac{l}{\mu c}\left(1 + \frac{\lambda E[X]}{(\mu c - \lambda E[X])}\right).$$

Last, $t_{i,n}$, $n > B$ can be bounded by $t_{i,n}^{(ub)}$ given by

$$t_{i,n}^{(ub)} = t_{i,B}^{(ub)} + b_{j-B}, \quad n = B+1, \cdots. \tag{13}$$

These expressions can be substituted into the following relation to obtain an upper bound on $E[T|X = i]$,

$$\begin{aligned} E[T|X = i] &\leq \sum_{n=0}^{\infty} \pi_n t_{i,n}^{(ub)}, \quad i = 1, \cdots, \\ &\leq \sum_{i=0}^{B} \pi_n t_{i,n}^{(ub)} + (1 - P[N \leq B]) t_{i,n}^{(ub)} \\ &\quad + b_j \left(E[N] + 1 - P[N \leq B] - \sum_{n=1}^{B} n\pi_n \right), \quad i = 1, \cdots. \end{aligned} \tag{14}$$

3.2.3 Approximate analysis of TS-PS

Let $T^{(lb)}$ and $T^{(ub)}$ denote the r.v.'s defined in the preceding sections that bound T from below and above. We use the following approximation for $E[T|X = i]$,

$$E[T^{(approx)}|X = i] = (E[T^{(lb)}|X = i] + E[T^{(ub)}|X = i])/2. \tag{15}$$

The accuracy of this approximation is high when the system load is low and/or when the parameter B takes a large value. We explore both of these effects in Table 1. Here we evaluate the upper and lower bounds on $E[T]$ for a system of 16 processors that process fork-join jobs containing exactly 16 tasks. The bounds are tabulated for different values of the processor utilization, $\rho = \lambda/\mu$ and for different values of B. We observe that sufficient accuracy is possible for processor utilizations up to .9 provided $B = 350$. In this case, the maximum error incurred by the approximation is 3.6% at $\rho = .9$ and less than .05% for $\rho \leq .8$. We shall use $B = 350$ throughout our studies.

ρ	B=50		B=100		B=200		B=350	
	lower	upper	lower	upper	lower	upper	lower	upper
.1	55.03	55.03	55.03	55.03	55.03	55.03	55.03	55.03
.2	56.68	56.41	56.68	56.68	56.68	56.68	56.68	56.68
.3	59.21	59.41	59.32	59.32	59.32	59.32	59.32	59.32
.4	62.95	63.91	63.47	63.47	63.47	63.47	63.47	63.47
.5	68.18	71.78	70.02	70.16	70.10	70.10	70.10	70.10
.6	75.17	87.01	80.60	81.70	81.17	81.17	81.17	81.17
.7	84.14	121.64	97.97	105.37	101.50	101.28	101.38	101.38
.8	95.20	225.95	126.68	175.40	142.94	148.56	144.88	145.04
.9	108.14	792.05	173.09	601.33	242.70	389.11	271.46	291.96

Table 1: Approximation Analysis

4 Comparison of Scheduling Policies

In this section we compare the performance of TS-PS, JS-PS, and FCFS. Specifically, we compare the mean job response time for different processor utilizations as we vary the number of processors and the job size. We also compare the performance of TS-PS and FCFS on a system that serves two classes of jobs: edit jobs and batch jobs. Edit jobs are assumed to consist of a single task whereas batch jobs consist of many tasks. Last, we consider the effects of partitioning the processors into two sets; one to serve edit jobs exclusively and the other to serve batch jobs exclusively. For this last study, we compare the performance of the partitioned system under TS-PS to one where the processors are available to all jobs under TS-PS.

4.1 Comparison of TS-PS, JS-PS, and FCFS

In this section we compare the TS-PS, JS-PS, and FCFS policies as a function of the processor utilization. In Figure 5 we plot the ratio of response times of TS-PS to JS-PS, and TS-PS to FCFS for two workloads. The workloads consist of jobs with a constant number of tasks that is equal to the number of processors, i.e., $X = 8, c = 8$ and $X = 16, c = 16$. The average task service time is taken to be $1/c$. From this figure we observe that FCFS provides uniformly better response over the two PS policies for all processor utilizations.

Furthermore, TS-PS gives lower response times than JS-PS for all processor utilizations. This is due to the fact that TS-PS takes advantage of the parallelism inherent in the fork-join job. The better performance exhibited by FCFS is due to the fact that TS-PS penalizes larger jobs, while no such penalty exists for FCFS (a more detailed discussion of this penalty phenomenon is given in the next section).

We also tested a workload consisting of two classes of jobs: edit jobs and batch jobs. Edit jobs consist of a single task and batch jobs consist of 16 tasks. Let f denote the fraction of jobs that are edit jobs. We considered three mixes, $f = .5, .95, .99$ operating on a system containing $c = 16$ processors. Figure 6 illustrates ratios of the mean job response time of TS-PS to FCFS as a function of the processor utilization ρ. We observe that the FCFS policy exhibits the best performance everywhere except when the utilization is high and the fraction of edit jobs is high ($f = .95, .99$). In this region TS-PS provides slightly lower response times.

This workload, ($f = .95, .99$), was chosen so as to increase the variability in the service job service times in an attempt to illustrate a setting in which TS-PS outperforms FCFS. It is surprising that the difference is so small. This is an indication that FCFS is a more robust policy on multiprocessors that execute parallel programs than it is in a system where jobs are executed serially.

From this figure we can also observe that TS-PS provides only slightly better service to edit jobs than FCFS, but significantly worse service to batch jobs.

4.2 Dependence on Number of Servers

In the last section we observed that TS-PS provides better performance than JS-PS for all of the examples. This appears to be at odds with observations that we noted in an earlier study [RTS87] on the performance of TS-PS and JS-PS in a *single processor* system. In a single processor system, JS-PS was shown to be uniformly better than TS-PS. This is due to the fact that in such a system there is no possibility for parallelism and the following occurs. Assume that there are 2 jobs, one with 1 task and one with 9 tasks. Then TS-PS

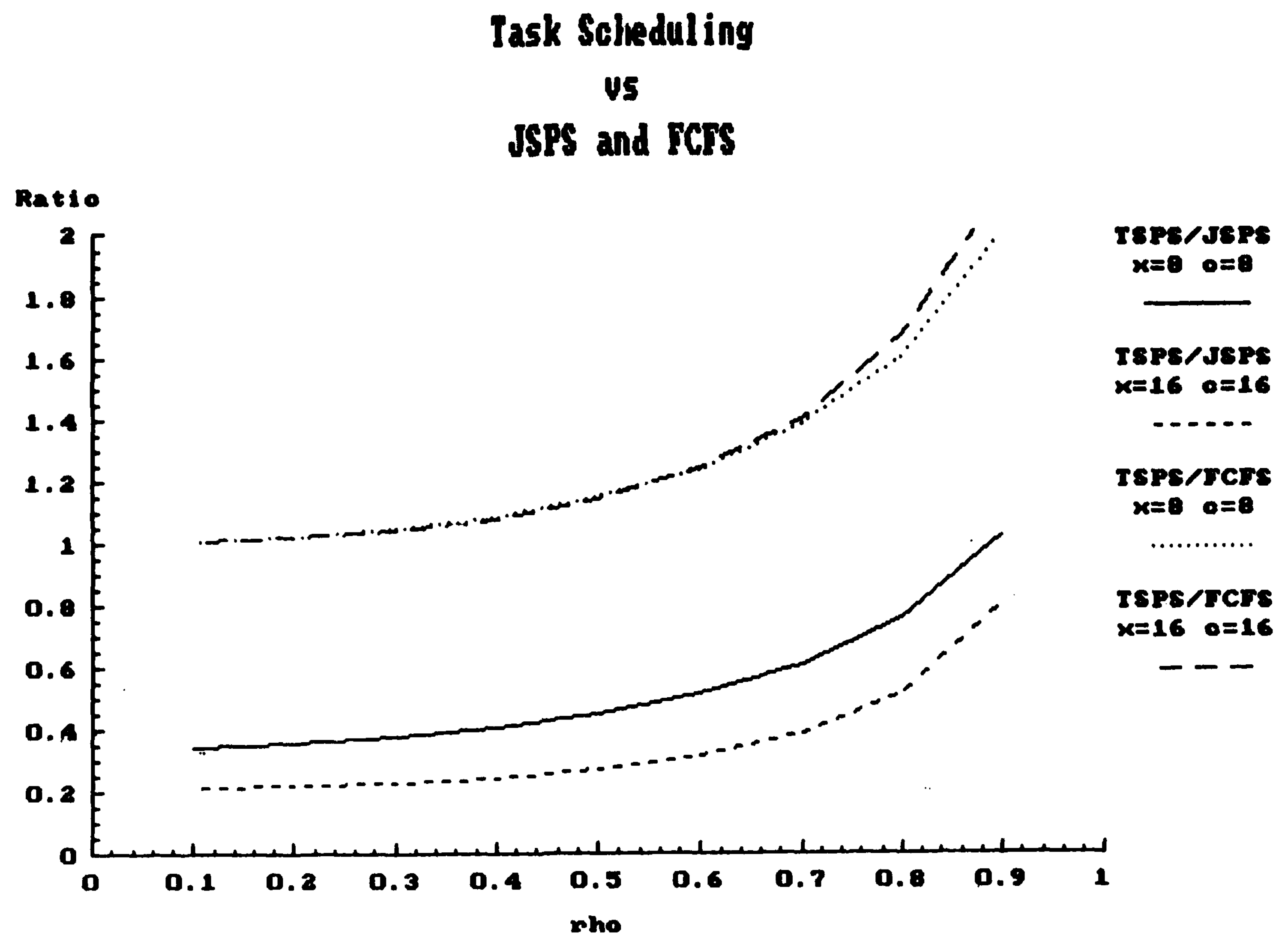
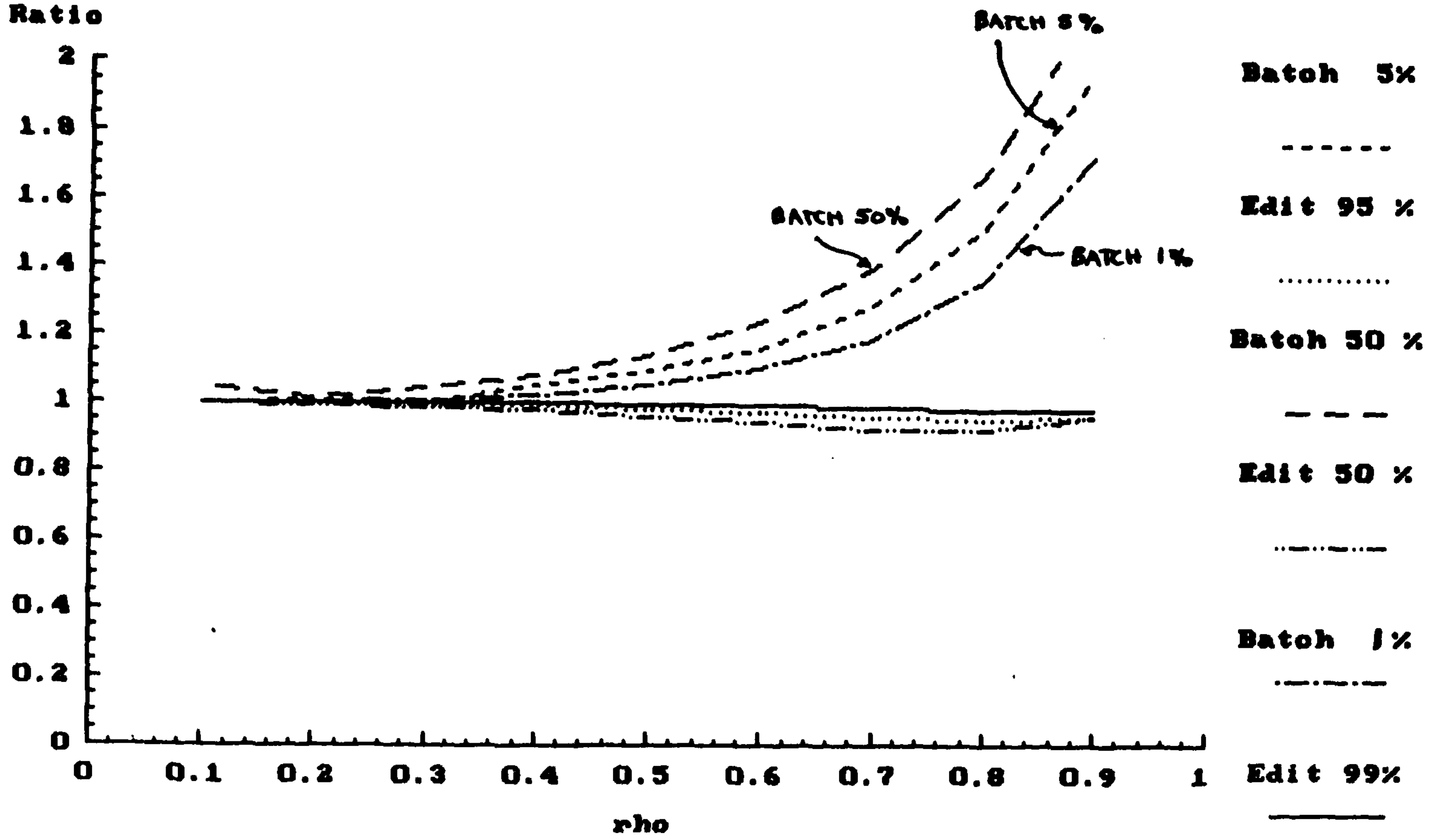


FIGURE 5

B=350

Batch and Edit
Jobs with
TSPS vs FCFS



o=16
x=16
B=350

FIGURE 6

gives the job with 9 tasks, 9/10 of the processor, and the job with a single task only 1/10 of the processor. However, JS-PS would give each job 1/2 of the processor. So on a single processor, TS-PS penalizes jobs with a small number of tasks. In a multiprocessor, there exists sufficient possibilities for parallelism so that this anomaly found in a single processor for TS-PS does not exist.

To study the effect of parallelism, we consider a workload of jobs consisting of 16 tasks and study the performance of TS-PS and JS-PS on systems with $c = 1, 2, 4, 8, 16, 32$ processors as a function of processor utilization. Figure 7 illustrates the results of this study plotting the response time ratios of TS-PS to JS-PS. We observe that TS-PS is always better than JS-PS in multiple processor systems when processor utilization is low. However, when the number of processors is small (< 8), there exists a utilization value, say ρ_0 such that system performance is better under JS-PS when $\rho > \rho_0$. This threshold is an increasing function of c the number of processors. This results because as the number of processors increases, the capability of sustaining parallel program execution under TS-PS increases.

4.3 Processor Partitioning

We now study the effect of dedicating a potion of the multiprocessor to each of the batch and edit classes. For edit jobs we assume that the computation time is small and equivalent to one task unit. Batch jobs are assumed to be large, consisting of fork-join tasks. The individual tasks from either class are assumed to have the same service requirements.

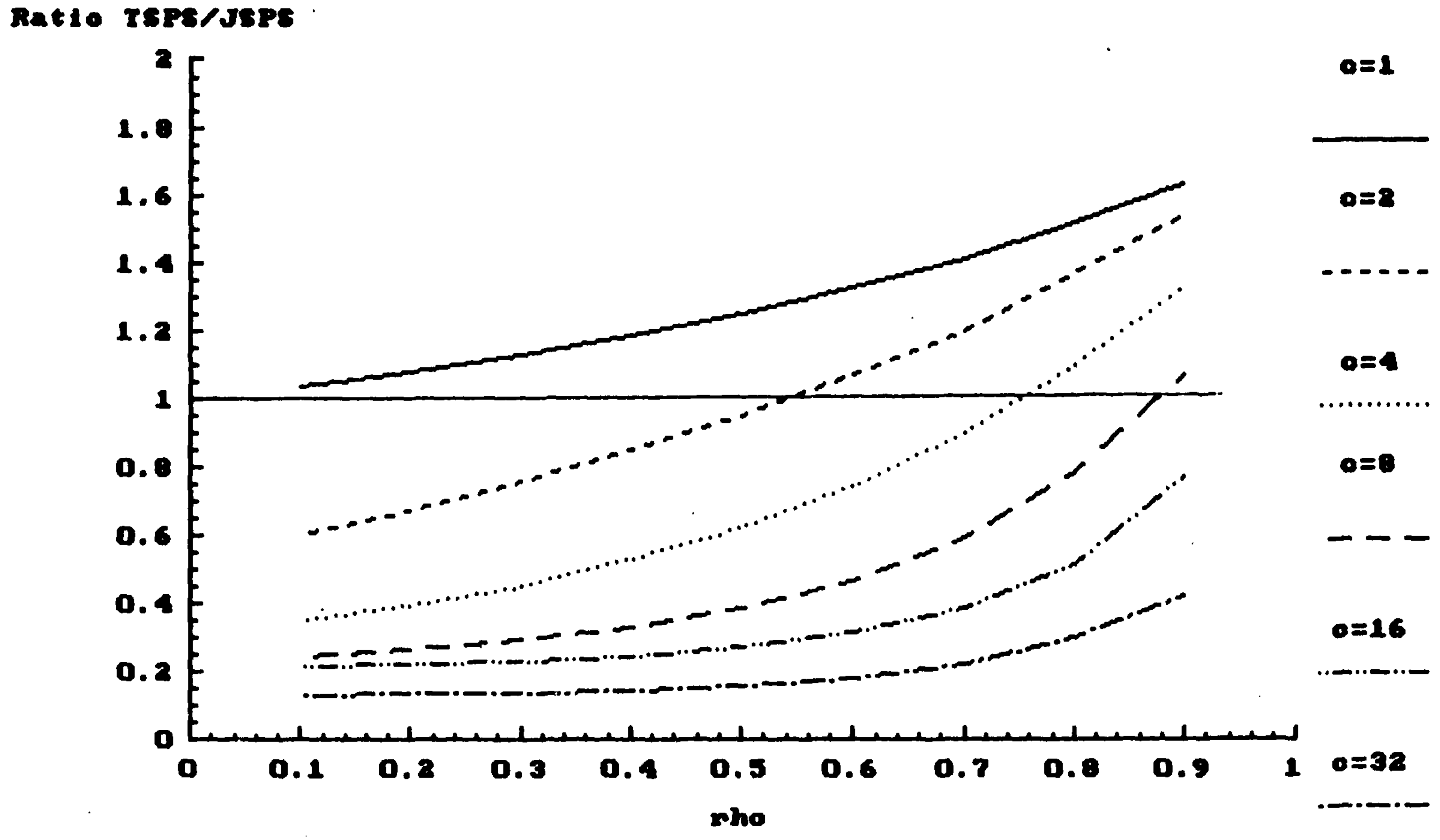
In order to examine the effect of statically dedicating a portion of the multiprocessor to each class, we compare the performance of a system composed of 16 servers where each server can run either class of job, to a partitioned system where some fraction of the processors are dedicated to each class. The combined system is composed of $c = 16$ servers. The partitioned system is composed of $c = 16$ servers such that K servers are dedicated to edit jobs and $c - K$ servers are dedicated to batch jobs. Our performance metric is the ratio of the response time of the partitioned system to that of the combined system.

In this experiment the independent parameter is the combined system utilization. Our first

TSPS/JSPS

VS

Number of Processors



x=16
B=350

FIGURE 7

experiment consists of an arrival of 50 percent edit jobs and 50 percent batch jobs. Note that this arrival pattern results in the total computation time of edit jobs to be 1/16 of batch jobs. The partitioned system is defined by K and the equivalent flow of jobs. We plot our results in (Figure 8).

We can observe several interesting phenomena from Figure 8. First, by dedicating only one server to the edit jobs, $K = 1$, both edit and batch jobs degrade. Thus, a poor partitioning choice negatively effects both classes of jobs. Second, improvements can be made in the edit jobs by allocating enough additional servers, $K = 2, 3$, to handle the computational load of edit jobs, but this is done at the expense of the batch jobs. This phenomena is especially striking at high utilizations.

As the relative arrival rate between edit and batch jobs increases, as show in (Figure 9) where the proportion of edit jobs is 95 percent, we see that more servers must be dedicated to edit jobs before the mean response time is decreased. Note that in this case the total computation time required by edit jobs is greater than needed by batch jobs. The result is that 9 of the 16 processors are required to reduce the edit job response times (see (Figure 9)). There are regions in the figure in which the performance of both jobs classes decrease, however, we observe no region in which both classes improve performance. This phenomena has also been reported in [NTT87] for FCFS scheduling.

Figure 10 reports the results when batch jobs are composed of 4 tasks and the workload contains 50% batch and 50% edit jobs. The results in this figure are similar to the 95% edit jobs and 5% batch job tests shown in the previous figure. The reason for this is that when batch jobs are fairly small, $x = 4$, and there are 50% edit jobs and 50% batch jobs in the workload, then the total computational requirements of edit jobs is high (as in the 95% test) for a given utilization. Therefore, edit jobs will saturate a small number of processors. Notice that only when the number of processors dedicated to editing reaches 4, does editing perform well.

Partitioning
at
50 %

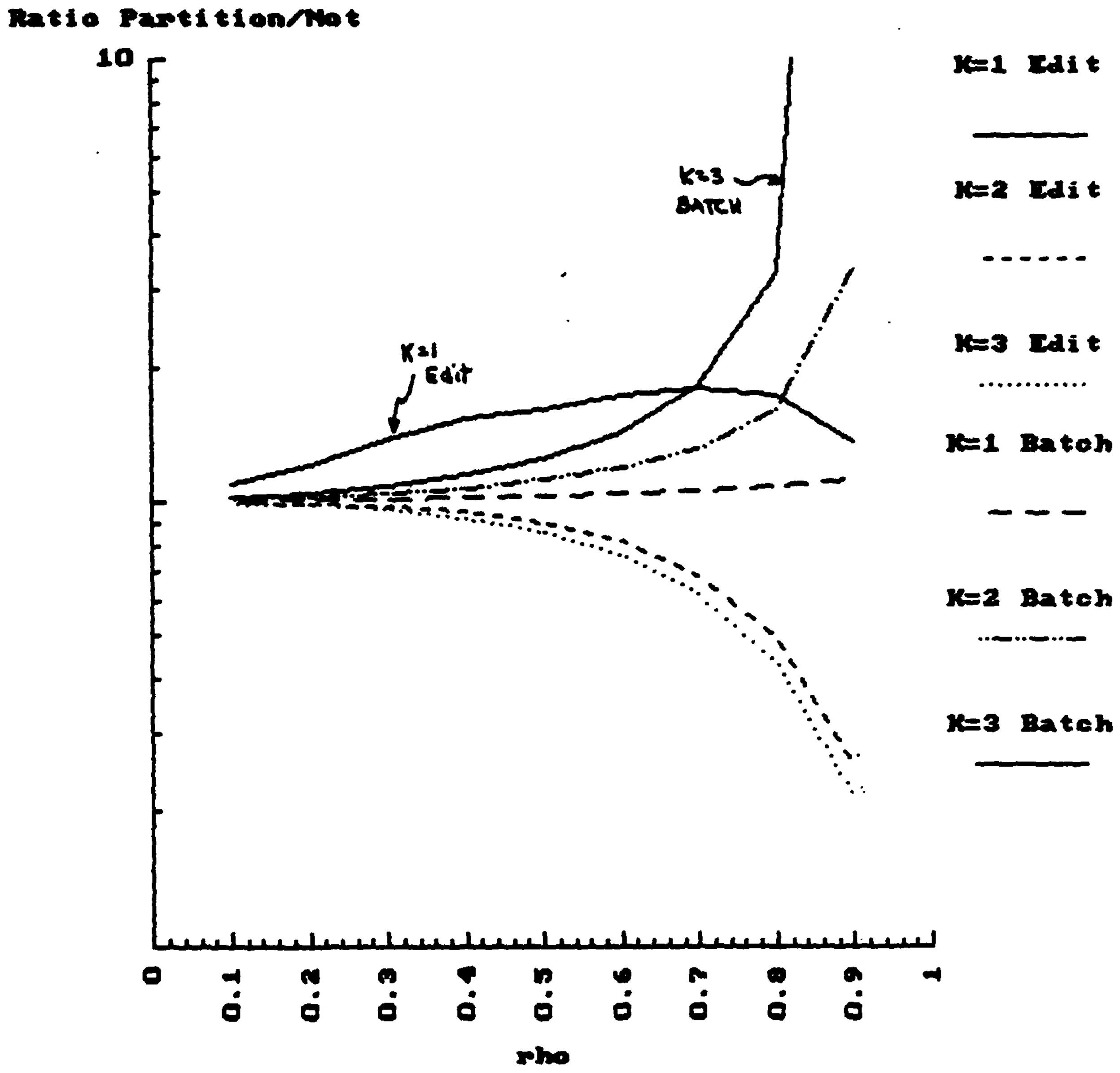
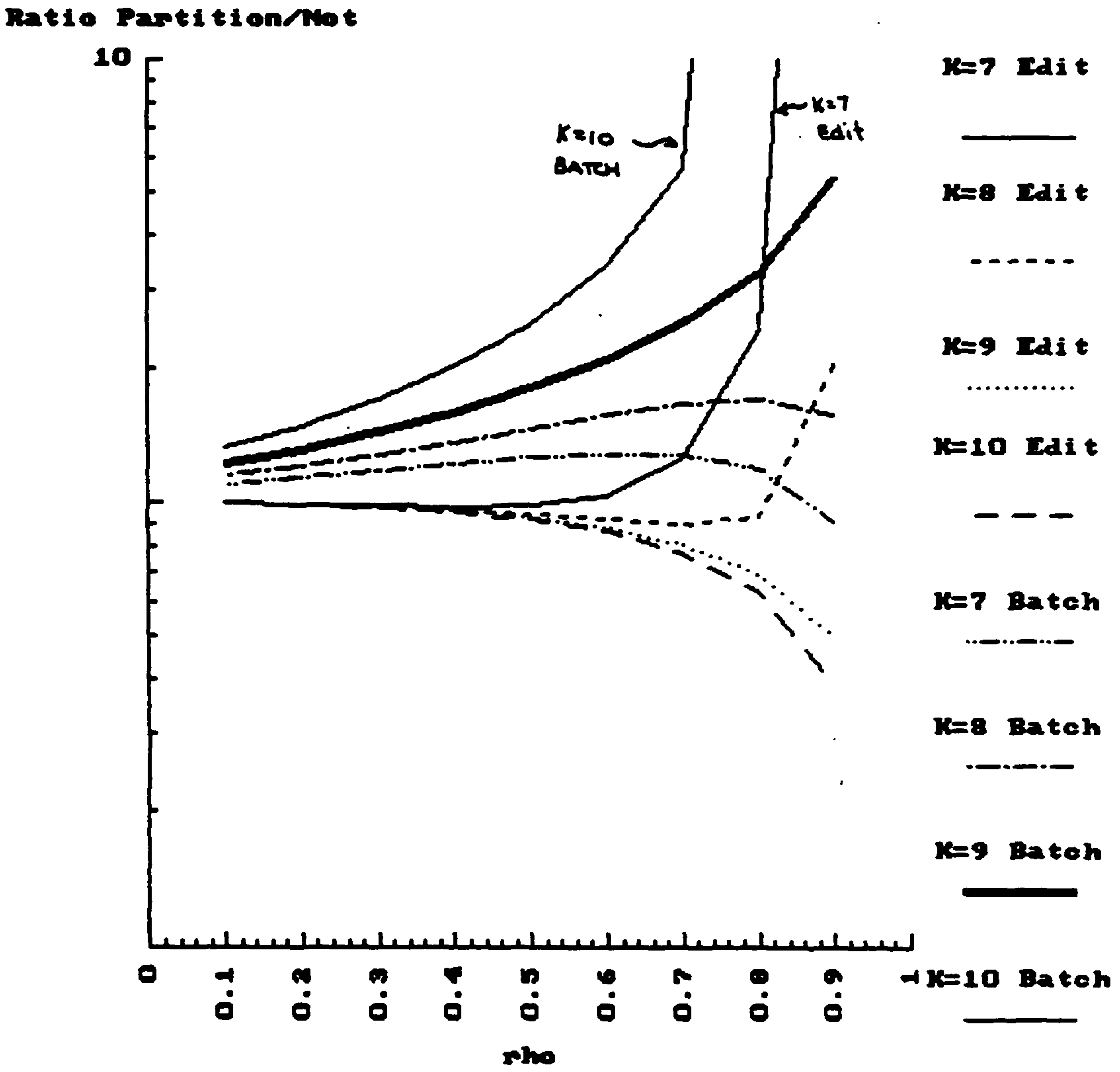


FIGURE 8

K=16
B=350

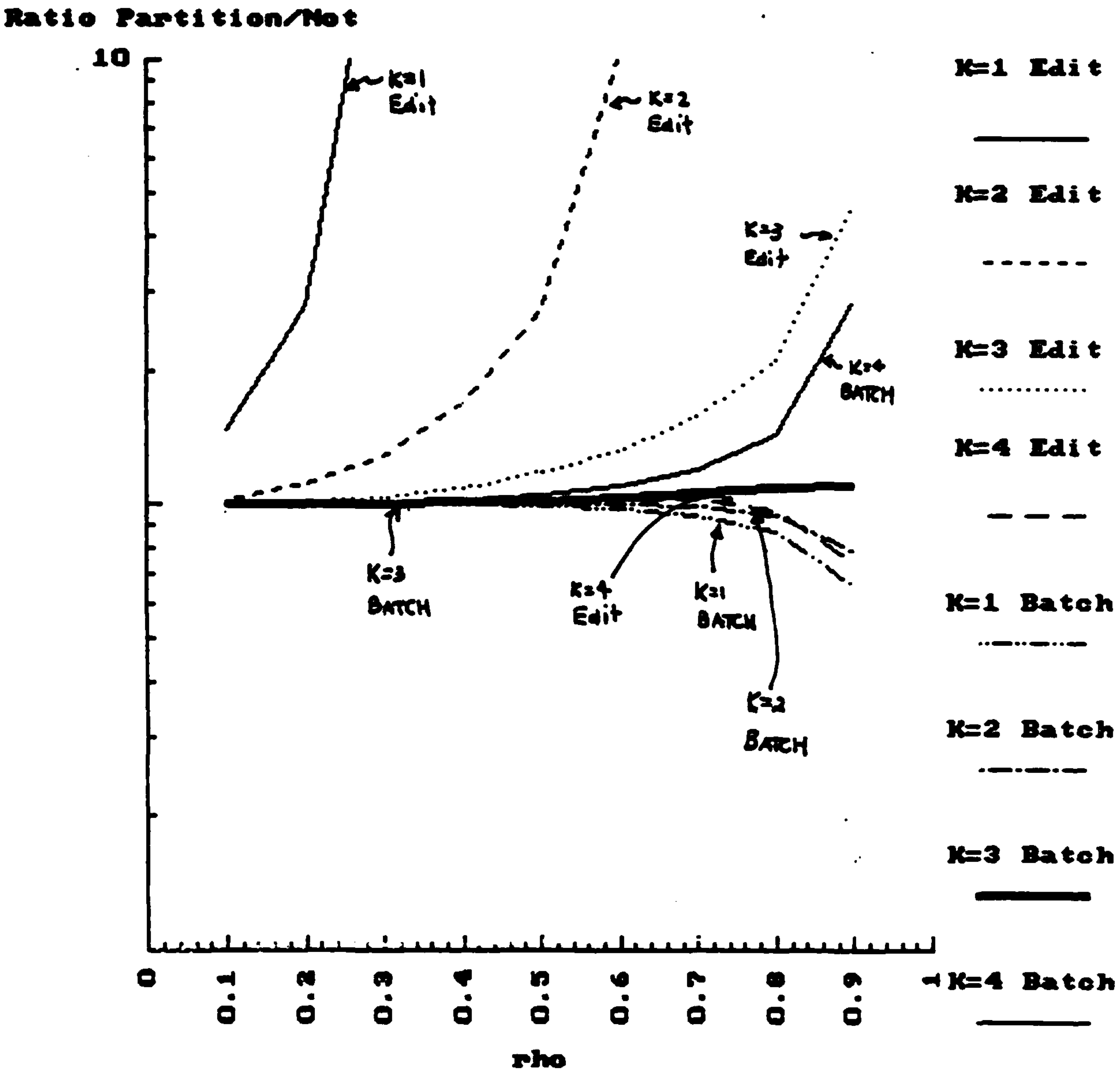
Partitioning
at
95 %



x=16
B=350

FIGURE 9

Partitioning at 50 %



x=4
B=350

FIGURE 10

5 Summary

We have analyzed fork-join programs as a $M^X/M/c$ queueing system. We have obtained am expression for the mean response time of a fork-join task under processor-sharing. Since our expression is not in closed form, but given as a set of recurrent equations, we have obtained expressions for both lower and upper bounds. Our bounds become tight as the number of states increase.

We have compared three scheduling approaches: TS-PS, JS-PS and FCFS. We have observed that in general FCFS out performs both TS-PS and JS-PS. Likewise, we have observed that TS-PS performs better than JS-PS unless that number of servers is small compared to the number of tasks.

We have considered the interesting problem of partitioning the system into two subsystems. Each subsystem is dedicated to one of two job classes: edit jobs and batch jobs. We determined several interesting results. When half the jobs are edit jobs and one server is dedicated for edit jobs exclusively, both classes experience an increase in response time. Improvements in edit jobs always cause a reduction in the performance of batch jobs in the partitioned system. This suggests that a parallel system should have a controllable boundary for processor partitioning.

References

[All78] Arnold Allen. *Probability, Statistics and Queueing Theory.* Academic Press, New York, New York, 1978.

[BM85] F. Baccelli and A. Makowski. Simple computable bounds for the fork-join queue. *Proc. Conf. Inform. Sci. Systems*, 1985.

[BMT87] F. Baccelli, W. Massey, and D. Towsley. Acyclic fork-join queueing networks. *submitted to JACM*, 1987.

[CT83] Chaudhry and Templeton. *First Course in Bulk Queues.* J. Wiley, New York, New York, 1983.

[GH76] Gross and Harris. *Introduction to Queueing Theory*. J. Wiley, New York, New York, 1776.

[Han75] P. Brinch Hansen. The programming language concurrent pascal. *IEEE Transaction on Software Engineering.*, 1, 1975.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.

[Kle64] Leonard Kleinrock. Analysis of a time-shared processor. *Naval Research Logistics Quarerly*, 11, 1964.

[Kle76] Leonard Kleinrock. *Queueing Systems Volume II: Computer Applications*. J. Wiley, New York, New York, 1976.

[NT85] R. Nelson and A.N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IBM Report RC11481*, 1985. to appear in IEEE Transactions on Computers.

[NTT87] R. Nelson, D. Towsley, and A. Tantawi. Performance analysis of parallel processing systems. *to be presented at SIGMETRICS '87*, 1987.

[Ost86] Anita Osterhaug. *Guide to Parallel Programming*. Sequent Computer Systems, Inc, Beaverton, Oregon, 1986.

[Pyl81] I.C. Pyle. *The Ada Programming Language*. Prentice-Hall International, London, 1981.

[RTS87] C. Gary Rommel, D. Towsley, and J. Stankovic. Ananlysis of fork-join jobs using processor-sharing. *Submitted to Operations Research*, 1987.

[Yao85] D. D. Yao. Some results for queues $M^X/M/c$ and $GI^X/G/c$. *Operations Research Letters*, 4, 1985.

Decentralized Decision Making For Task Reallocation In A Hard Real Time System

John A. Stankovic[1]
Dept. of Computer and Information Science
University of Massachusetts

14 August 1986

Accepted for publication in IEEE TRANS on Computers, being revised.

[1] This work was begun while the author was on sabbatical at CMU and working on the Archons project. It was supported by ONR under contracts 048-716/3-22-85, NOOO14-84-K0734, and RADC Contract 93921-39.

Abstract

We have developed and analyzed a decentralized task reallocation algorithm for hard real-time systems. The main properties of the algorithm are that it is decentralized and reliable, it specifically considers deadlines of tasks, it attempts to utilize all the nodes of a distributed system to achieve its objective, it is fast, it handles tasks in priority order, and it separates policy and mechanism. An extensive performance analysis of the algorithm via simulation has shown that it is quite effective in performing reallocations, and is significantly better than a centralized approach.

Keywords: Centralized Algorithm, Cooperation, Decentralized Algorithm, Decision Making, Decentralized Decision Making, Hard Real-Time, Reallocation, Real-Time, Reconfiguration, Scheduling, Simulation Study.

1 Introduction

Distributed computer systems potentially provide significant advantages including good performance, high reliability, significant resource sharing, and extensibility. One particular class of distributed computer systems contains hosts that are highly integrated and being used for a *single* application such as process control, control of a nuclear power plant, or control of a submarine, aircraft carrier or space shuttle [1]. These dedicated distributed systems are often associated with real-time environments in which there is an especially strong requirement for meeting deadlines, for reliability and for continued operation in the presence of failures. In this paper we are concerned with one aspect of reliability, that is, reallocating real-time tasks throughout a distributed system after a single host failure. We show that distributed decision making by a system-wide reallocation algorithm is a better approach than a centralized approach with backup.

Decentralized decision making can be a complicated process fraught with dangers such as instabilities and long delays. On the other hand, it also provides potential benefits of increased performance, greater reliability and easier extensibility than a centralized decision making process. Where the truth lies for a specific situation is dependent on many factors including the requirements, the environment, and the quality of the decision making process. We are interested in investigating decentralized decision making for task reallocation after a host failure in a distributed hard real time system where hard refers to the fact that tasks have periodic constraints or deadlines. We have performed such an investigation by developing both a decentralized algorithm and a centralized algorithm, comparing each of them with respect to performance, reliability and extensibility. We conclude that the decentralized algorithm does have significant advantages over the centralized algorithm.

The decentralized reallocation algorithm described in this paper illustrates the types of tradeoffs a designer must make when developing a decentralized algorithm. While it is the specific intent of a decentralized algorithm to have multiple, physically distributed agents of the algorithm negotiate and/or cooperate with each other, such interaction has a cost. The proper balance between making a decision quickly using the current available information versus accepting an additional delay and cost of obtaining new information and/or coming to a consensus with other agents of the decentralized algorithm must be achieved. For the reallocation algorithm in a hard real time system, the tradeoff bias is on the side of performing actions quickly and in a highly autonomous fashion. The main contributions of this paper are the development of an efficient decentralized task reallocation algorithm tailored to real-time constraints, and its analysis. We also specifically address the tradeoff between execution time speed of the algorithm and cooperation.

Section 2 presents basic background information about hard real time systems, and lists our assumptions. Section 3 presents the decentralized reallocation algorithm itself. Section 4 describes the simulation program, the centralized algorithm and contains the simulation results on the performance analysis of both the centralized and decentralized algorithms. Section 5 discusses the tradeoffs made in this algorithm and the need for a meta level controller. Section 6 summarizes the conclusions of the paper.

2 Hard Real-Time Systems

2.1 Architecture of the System

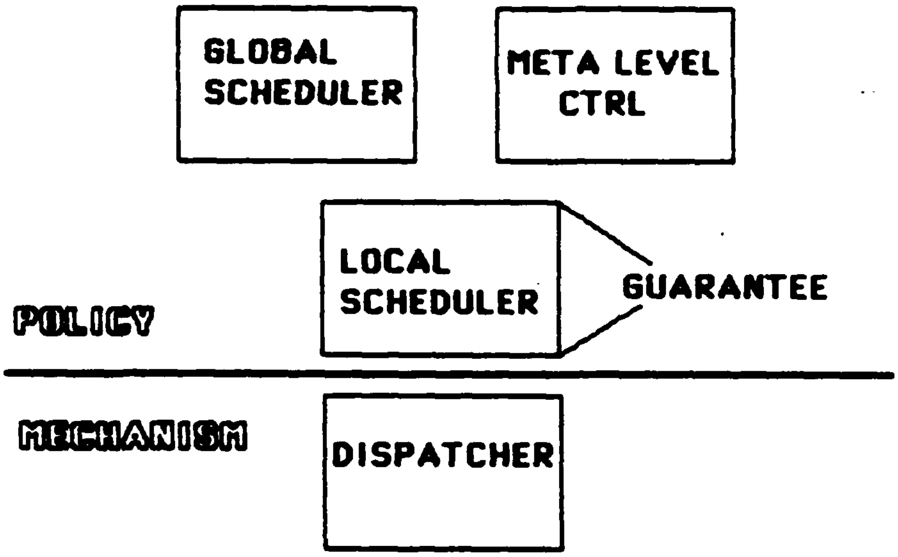
A Hard Real-time system [2][12] is defined as a system in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. In such a system we expect that many of the tasks, not just the I/O tasks, have real-time constraints such as start times, deadlines or periodic constraints. Further, these real-time constraints must be met, else there are potentially catastrophic consequences. Examples of this type of real-time system are command and control systems, process control systems, flight control systems, the space shuttle avionics system, and automated factories. In each of these types of real-time systems one typically finds multiple levels of timing constraints. For example, processing data from a sensor might need to occur in the microsecond or millisecond range, while consistent updates of replicated files might need to occur in the seconds range, and the movement of material in an automated factory might have deadlines in the range of minutes or hours. Our overall approach assumes that tasks with tight time constraints are dealt with by using preallocated devices, processors or time slots. The tasks with intermediate or long range timing constraints are dealt with in a dynamic manner, and it is these tasks that are the subject of possible reallocation.

We assume that the system is physically distributed and composed of pools of special purpose and identical general purpose nodes networked via a shared bus, or collections of busses connected via bridges. The reallocation algorithm is concerned only with the general purpose processors. Each node is a multiprocessor where one or more processors are dedicated system processors, and one or more are application processors. The system processors run the operating system which includes programs such as the distributed scheduling and distributed reallocation algorithms. The scheduling algorithm separates policy from mechanism and is composed of 4 modules (Figure 1). At the lowest level there exists a dispatcher. The dispatcher is the mechanism of the scheduler and it simply removes the next task from a system task table (STT) that contains all guaranteed tasks already arranged in the proper order. The second module is a local scheduler. The local scheduler is responsible for locally *guaranteeing* that a new task can make its deadline, and for ordering the tasks properly in the STT. The third module is the global scheduler which attempts to find a site for execution for any task that cannot be locally guaranteed. The final module is a meta level controller which has the responsibility of adapting various parameters by noticing significant changes in the environment and serving as the user interface. The meta level controller may not always exist and is only briefly discussed in this paper.

The basic notion and properties of the guarantee have been developed elsewhere [6][7][8][10] and has the following characteristics,

- there is a separation of dispatching and guarantee allowing these system functions to run in parallel; the dispatcher is always working with a set of tasks which have been validated to make their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks,

SCHEDULING ALGORITHM COMPONENTS



GLOBAL SCHEDULER

META LEVEL CTRL

LOCAL SCHEDULER

GUARANTEE

POLICY

MECHANISM

DISPATCHER

- FLEXIBILITY
- CONFIDENCE IN RT CONSTRAINTS BEING MET
- LOWER COST (MORE AUTOMATIC)

FIGURE 1: SCHEDULING ALGORITHM

- by performing the guarantee calculation when a task arrives there may be time to reallocate the task on another host of the system via the global module of the scheduling algorithm,

- the guarantee can employ different strategies as a function of the deadline of the incoming task,

- within this approach there is notion of still possibly making the deadline even if the task is not guaranteed, that is, if a task is not guaranteed it receives any idle cycles and in parallel there is an attempt to get the task guaranteed on another host of the system subject to location dependent constraints; an alternative is to run a substitute task and/or error handler early, rather than only after a deadline is missed,

- some real-time systems assign fixed size slots to tasks based on their worst case execution times, we guarantee based on worst case times but any unused cpu cycles are automatically reclaimed, and

- in general, the guarantee is subject to computation time, deadline or period, resource requirements, priority, precedence constraints, I/O requirements, and atomic transaction requirements.

Note that the reallocation algorithm will make use of the guarantee routine as well as the global scheduler, since reallocation is essentially scheduling at the time of failure. It can be considered scheduling burst arrivals where the burst of tasks is defined to be those tasks on the failed processor.

In summary, we assume that there exists a system with the following requirements:

- multiple, physically distributed hosts where each host is a multi-processor composed of at least one processor dedicated to system tasks and one dedicated to application tasks (Figure 2),

- tasks have deadlines (periodic and nonperiodic),

- the set of tasks active at a site is very dynamic,

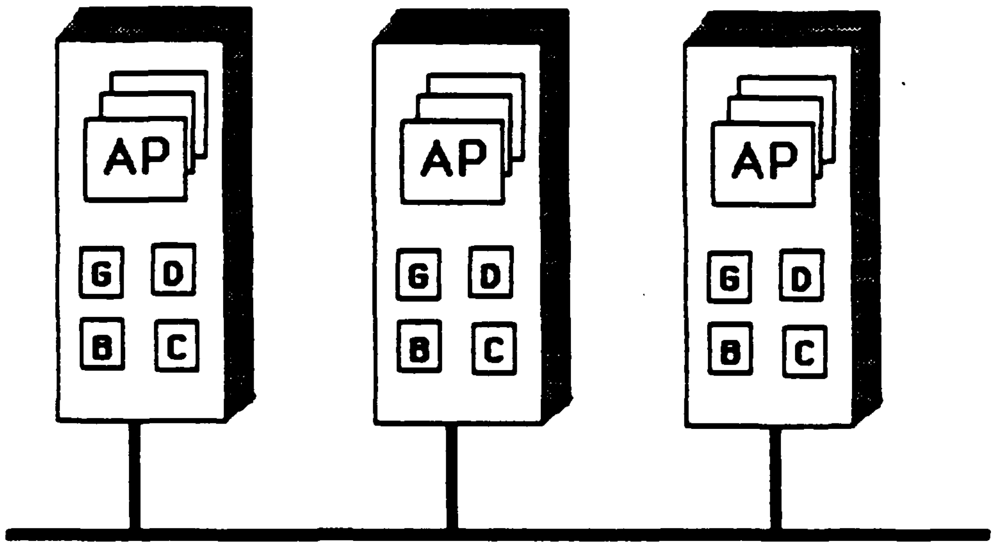
- when one or more processors fail-stop it is critical to reallocate as quickly as possible with the reallocation task itself being subject to a deadline,

- the reallocation process itself should be reliable,

- the overhead for the reallocation process should be as little as possible during normal operation of the system,

- active tasks should continue to execute, if feasible, during the reallocation itself,

- there exists a wide priority range among the tasks,



AP -> APPLICATION PROCESSORS

G -> GUARANTEE ROUTINE

D -> DISPATCHER

B -> BIDDER AND FOCUSED ADDR.

C -> COMMUNICATIONS PROC.

FIGURE 2: MULTIPROCESSOR NODES

- highest priority tasks are very important and under failure it is necessary to attempt to abort as many low priority tasks as necessary to get one or more high priority tasks to run,
- reallocation routine has priority X (high), but there are some routines of higher priority, and
- on the average, the cost of communication, of transferring tasks, and of I/O is proportionally less than computation times of tasks; tasks arrive with reasonable laxities and arrive early enough for it to make sense to attempt to use other sites in the network for its execution.

The above set of requirements are quite demanding and are typical for developing large and reliable command and control systems. We hypothesize that they can be achieved to the greatest extent via decentralized control. Our first step is to develop a decentralized algorithm that can meet these requirements. We then compare this decentralized algorithm to a centralized algorithm. In the next subsection we more fully describe the characteristics of tasks in hard real-time systems.

2.2 Characteristics of Tasks

The task is the dispatchable unit of computation in the system and is non-preemptable. We assume that tasks are independent. When a task is invoked it specifies:

- its deadline or period where period is defined to mean that a given task must execute once per period P until manually terminated,
- its worst case computation time,
- its priority (identifies the criticality of a task; independent of deadline),
- a reallocation factor,
- a replication factor for active copies,
- a replication factor for passive copies (all copies of a particular piece of code are identical - updates to code are performed as an atomic action across all active and passive copies), and
- its type
  - single task
  - voting task
  - decentralized task.

A single task is one that has one active copy and executes in one location only, thereby using a minimum of resources. However, if the host upon which this task is executing fails, the system attempts to reallocate it subject to the task and system characteristics at the time of the failure. The replication factor for a single task is one. The reallocation factor is ≥ 1 and it represents the number of other hosts which must be responsible for reallocating this task. In addition to having ≥ 1 host responsible for reallocating the task, there must exist ≥ 1 passive copies of the task; this is the replication factor for passive copies. The copies of the task itself may or may not be at the hosts responsible for deciding where to reallocate the task, and this must be taken into account when making a reallocation decision.

A task may request that it be instantiated n times and run in parallel, typically for voting on the outputs. Such a task, called a voting task, is inherently more reliable, but consumes more system resources. In addition, if a host upon which one of the replicated copies is executing fails, then the system attempts to reallocate that failed instance of the task unless there are still enough copies for adequate voting. If there are enough copies, then (for efficiency) the task instantiationon the failed host need not be reallocated. The number of sites responsible for reallocation is the reallocation factor. Passive copies of this task also exist, but typically the number is less than for a single task because there are other active copies available. For efficiency considerations we make use of the fact that there are multiple active copies.

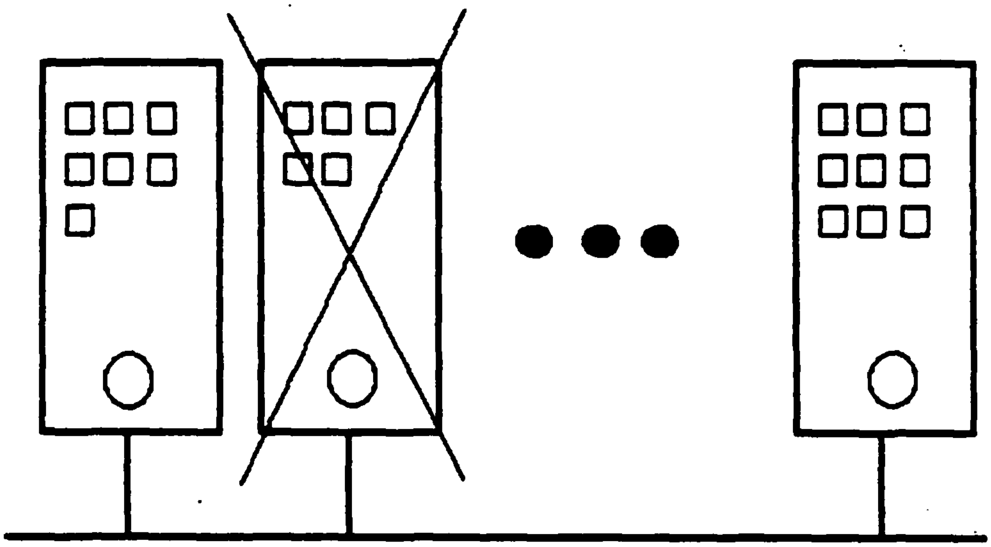
A decentralized task is a collection of rtasks (replicated tasks[1]) which cooperate in a peer relationship to achieve some system-wide goal. There is no voting on the results of each computation. Many of the system server functions such as the naming server, the scheduling algorithm, the reallocation algorithm, the directory server, the resource manager, etc. are decentralized tasks. Such tasks have a minimum replication factor specified by the replication factor for active copies. Depending on the function themselves, it is possible for new active copies of a decentralized task to be instantiated to offload work, e.g., as was done in the Medusa utilities. If failures result in a particular decentralized task dropping below the minimum replication factor, then reallocation is performed, else this instantiation of the rtask is not reallocated (again improving the performance of the reallocation algorithm itself because it might have less work to perform). Work that this rtask was performing must be reassigned to some other rtask of this function.

2.3 The Reallocation Problem

The goal of this work is to develop a good task reallocation algorithm for hard real-time systems where reliability and continued performance in the presence of failures is of utmost importance. Simply letting the tasks on the failed processor be lost it not acceptable. We are interested in determining the conditions under which a decentralized algorithm (Figure 3), or a centralized algorithm (Figure 4) is better. In any case, the algorithm must execute fast because it is dealing with hard deadlines, must be reliable, and must perform good reallocations, i.e., as many of the tasks on the failed processor as possible must be reallocated

[1] Informally referred to as agents in the Introduction.

THE TASK REALLOCATION PROBLEM

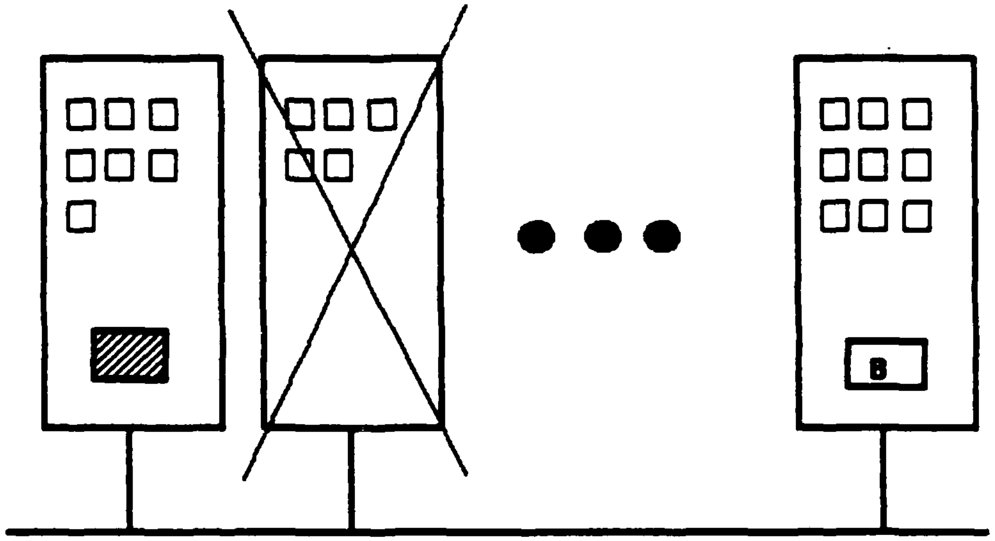


□ APPLICATION TASKS

○ DECENTRALIZED REALLOCATION ALGORITHM

FIGURE 3: DECENTRALIZED REALLOCATION

THE TASK REALLOCATION PROBLEM



□ APPLICATION TASKS

▨ CENTRALIZED REALLOCATION ALGORITHM

B BACKUP

FIGURE 4: CENTRALIZED REALLOCATION

and make their deadlines. When the reallocation algorithm decides on a new location for a task, that task must then be guaranteed at that site. If it is guaranteed, then it will make its deadline. If not, for the purposes of this paper, we consider that it fails to meet its deadline. However, in practice, our scheduling algorithm which is based on the guarantee routine would let this task use idle cycles and so it may still may make its deadline - it just won't be guaranteed to do so. Such idle cycles are attained when a guaranteed task executes for a time less than its worst case time (very common), since it was guaranteed with respect to the worst case time.

The reallocation problem is actually composed of multiple subproblems:

- detect the failure,
- notify the reallocation managers (algorithm) at each site,
- stop certain tasks and task interactions,
- decide on the new allocation of tasks,
- perform actual movement of tasks,
- recover, and
- restart tasks

In this paper we concentrate on the *decide on the new allocation of tasks* part of the overall problem, and assume that the other parts of the problem are handled in some effective manner.

To our knowledge, the current literature has not reported on a decentralized reallocation algorithm in a hard real-time environment. A centralized algorithm was reported in [11], but it was not evaluated by simulation and was aimed at a considering resource requirements such as ports and memory constraints. Since reallocation is so similar to scheduling, some of the hard real-time scheduling literature is related including [3][4][5] and [9]. However, this work does not address the specific issues related to reallocation, as is done in this paper.

3 The Decentralized Reallocation Algorithm

Consider that at some time T, the real-time distributed system is composed of M tasks assigned across N hosts, typically $M >> N$. Also consider that a single host fails. At this time the decentralized task reallocation algorithm is invoked to reassign tasks of the failed host. In this section we first describe the overall structure of the decentralized reallocation algorithm discussing various options and provide motivation for our decisions on these options. We then present the pseudo code for the actual algorithm.

- In the most general case, when a task is assigned to a site, it is done with respect to its own requirements and the host characteristics. Its own requirements include its deadline, requirements for ports, memory, and bus bandwidth, and its precedence requirements, its need for data and I/O taking into account its possible contention over those resources. Host characteristics include its current state (load on resources) and its speed, amount of memory, I/O capability, and any special devices capability. A local guarantee routine (a local system-level scheduler) [6][7] is responsible for determining whether a task can be assigned to this particular site (although it may require global or subnet information), i.e., the local scheduler determines whether the task can obtain the necessary resources at this host at the correct time in order to meet its deadline. In the simulations we assume that resources are available to a task when it executes.

- When a task is to be assigned to a site, buddy sites (number is based on the reallocation factor n) are also chosen, and the assignment and buddy information needs to be accomplished as an atomic action. A fundamental point is that some site (or for reliability n sites) must know that a given task has been assigned to execute at a given site. These buddy sites are responsible for reallocating the original task if the processor on which the original task was assigned fails. The responsible sites can be chosen in any number of ways and once chosen the decision making of where the task is to be reallocated can also be performed in any number of ways. Guidelines for choosing the responsible sites and the decision making strategy are as follows:

    - Random: For the given task under consideration, determine the set of feasible sites to which it might be reallocated. This is done based on unique resource requirements. Then choose n sites at random (or less than n if the number of feasible sites is less than n). As each site is chosen create a virtual chain (order) which acts as the mechanism for the decision making strategy. That is, on failure, the first buddy site in the virtual chain makes the decision without negotiating, consensus or cooperation of any kind with the others in the virtual chain. The other sites in the virtual chain are only activated one at a time, if there are subsequent failures.

      This strategy is meant to make decisions quickly and efficiently and to partition the workload when a failure occurs. This scheme is very quick at determining the buddy sites (minimizing overhead during normal operation) and if we have a broadcast subnet it will only cost 1 message and one commit operation. One can attempt to be much smarter about assigning buddies based on (1) resources available at a site, or (2) on trying to guarantee that no one site will be responsible for too many tasks from the same site. However, due to the dynamic nature of our hypothesized system, there is no guarantee that the initial assignment of a buddy will be appropriate at the time of a failure. Therefore, it seems that it is not appropriate to try to make assignments based on resource availability. It is reasonable to attempt to insure that a given site is not responsible for too many tasks from the same site. If it were, then it would have too much to do if a failure of that site occurred. The random scheme we outlined above should accomplish this most of the time. We could also add (but do not) a simple check in which a site periodically looks at the buddies assigned to it, and if too many are from one site it can reassign some of them. The reallocation decision itself is made at one buddy site and we attempt to minimize the communication delay experienced between the up to n buddy sites. However, the buddy that has responsiblity for the decision on where to reallocate may still communicate with the decentralized scheduling algorithm at a remote site.

Note that because of the random assignments of buddies, when a site fails the primary buddy sites will be able to proceed in parallel, each making reallocation decisions for some subset (possibly null) of the tasks that were active at the failed site. This is decentralized but without direct cooperation. However, because the decentralized reallocation interacts with a decentralized scheduling algorithm, there is cooperation being achieved *implicitly* via the scheduling algorithm.

- When a non-periodic task completes, or if a periodic task is terminated, then we must deactivate the buddies. This cost is one reliable broadcast message. To prevent race conditions the deactivation of a task and its buddies must be done as an atomic action.

- When a site failure is detected, broadcast this fact. This broadcast must be reliable. A failure of a host results in other hosts attempting to reallocate all of the active tasks at this site, except possibly the rtasks of decentralized tasks which still have the number of instances $\geq$ the minimum replication factor, and except possibly the replicated voting tasks which have enough members remaining to still provide a majority vote. In addition, it may be necessary to update the virtual chain of buddies. Two philosophies are possible here. One can consider the original reallocation factor static. That is, if $n = 5$, then after a failure, it is reasonable to now be at level $n = 4$. The other philosophy is to dynamically attempt to retain the $n = 5$ level. We decided on the first strategy. (Actually, the same is true for the replication factor and we use the same philosophy there too.) In the static strategy that we chose , the original virtual chain may have to be modified on a failure. For example, if the site chosen to execute the task is the primary buddy site, then it can no longer be the primary buddy. The second buddy now becomes the primary via a broadcast message (other sites must know too). Further, if one of the sites in the virtual chain fails, then it is clear that this site is no longer in the virtual chain. This condition is easily handled by having each site keep track of all failed sites.

- Each site, upon receiving information about a processor failure, attempts to activate at this site all buddies of tasks that were executing on the failed processor and that it has primary responsibility for. It does this in priority order. If all tasks that it is responsible for can run locally, then the algorithm invocation at this site is done. This strategy is again an attempt to make *good enough* decisions quickly.

- If one or more buddy tasks cannot be handled locally, then find a new active site to execute that task, and remain the primary buddy. To find a new site for a task, a site always works with the set of tasks that it knows needs to be reallocated. This distributes the reallocation workload among all processors, but could be subject to stability problems. Finding a new active site can be accomplished in many ways. At this point of the algorithm, it does make sense to attempt to find a good location for the task based on resource availability so that the task being reallocated has a chance of making its deadline. A general point to make is that we also need to eliminate from further consideration, those tasks which are too close to their deadlines. Trying to process these tasks will only increase costs and not benefits. Here we treat periodic tasks differently than non-periodic tasks. For example, it may be reasonable to accept cancelling the next m instances of a periodic task so that a good location for it may be found, and from that point on it will make its recurring deadline. This strategy cannot work for the non-periodic tasks. We use the following strategies in combination (here we concentrate on describing what is done for the non-periodic tasks):

- Focussed Addressing (FA): In FA a task is transmitted directly to some site. If a site that is attempting to reallocate a task happens to have recent (good) information such as that another site is very likely to be able to execute this task, then it directly transmits the task to this site without incurring any bidding overhead and delays. FA has been shown to work well under these circumstances [8]. Overreaction of the more heavily loaded sites of the network to a lightly loaded site must be prevented. As part of this phase of the algorithm (and depending on the cost benefit tradeoff), we need to modify state information about the site chosen as a FA site, estimate what other sites will do, and possibly inform others of our decision.
- Bidding: Bidding is performed when a site does not have good enough information (e.g., too old) or all the information it does have, does not identify a good destination. Here we use bidding tuned to real-time constraints [10]. Even after bidding, a requesting site may not receive any good bids. This could then require preemption. This may be too expensive in many cases, and may indicate a need for a *one* round of messages solution. For example, each site might exchange their current state at the time of failure and then make a decision - period. Or, each site may exchange their state only after they all complete their local decision making. This would reduce communication costs, but not benefit from any parallelism. There are stability issues regarding bidding too. Bidding can be reduced by requesting bids on the collection of tasks still not guaranteed when bidding is entered. Various heuristics must be developed to handle multiple simultaneous bid requests from one site. As an example, suppose site 3 has 4 tasks still unassigned when the bidding is entered as part of its portion of the reallocation algorithm. Site 3 then issues a RFB for all 4 tasks ordering them by priority. The RFB is then a variable length message, and, of course, contains the D, C, resource requirements, etc. of each task. In the worst case, a responding site would have to make bids on every combination of the 4 tasks, i.e., bid on task 1 alone, on task 2 alone, ..., on task 1 and 2, on task 1 and 3, ..., etc. This approach is too costly. The first heuristic that comes to mind is to have the responding site bid on each alone (i.e., as if the

responding site were to bid on that task by itself and none of the others, and on an accumulation of tasks from the top down (i.e., from high to low priority). For example, in the above example with 4 tasks (ordered by priority), the responding site would bid on task 1,2,3,4 each alone, as well as on tasks 1 and 2, 1, 2 and 3, and 1,2,3,4 all together. All this information could be returned in one bid message. This heuristic reduces the combinations of tasks to be bid on and does it using priority as an important measure. The site receiving bids of this nature would be responsible for making some ultimate choice based on all received bids. The simulation implements the standard bidding technique of addressing one task at a time.

- Preempt Locally: When it was determined that a primary buddy task, suddenly activated, cannot be handled locally, this calculation was done without considering preemption. This same task might be schedulable locally if we preempt lower priority tasks at this site, i.e., we remove lower priority tasks from the dispatcher list (STT). In other words, preemption refers to cancelling lower priority ready tasks, and not to interrupting tasks in execution. It is a policy decision on whether to first attempt FA, then bidding then preemption locally. Other orders of execution might be just as appropriate. It is possible to make decisons on the order of execution dynamically based on the system state or on policy. For example, host A has a task B to reallocate and its view of the system is that all hosts are very busy. In this case the algorithm might try to preempt locally as the first step.
- Preempt Globally: Given that a task cannot execute locally, then the criterion might be to attempt to schedule it by preempting the lower priority tasks system-wide that would free enough resources for this task to execute by its deadline. This scheme would probably require significant overhead if done in a decentralized manner. It may be more feasible with the approach briefly mentioned above of exchanging a round of messages or with the method indicated below. See [11] for a technique for global preemption. Since our simulations indicate that there is little need for global preemption, we do not describe a technique here.
- The first time a task cannot be assigned locally, we expect that there is a preferred order for these four strategies, e.g., FA, Bidding, Preempt locally and then preempt globally. However, it may be a good idea to dynamically alter this order, e.g., begin with FA but choose between the next three based on this site's view of the network. If the network looks busy then do Preemtp locally, and/or preempt globally before bidding. If the network is not too busy then do bidding before any preempting. Whatever strategy is used for the first attempt at reallocation this task, and this task is moved, then the subsequent site strategy need not be the same because of the additional delay already experienced with making this wrong choice. The subsequent site strategy might best be Preempt locally, FA, bidding (or null), then preempt globally. Preempt globally is a difficult problem because network-wide information about priority is needed. Consider that when preempting globally a RFB is sent with the characteristics of the task (D and C) and its priority. A

returning bid would include ordered pairs, (priority, resources available if tasks of this priority are preempted), up to the priority minus 1 of the task being bid on.

- When a task arrives at a new site, the local scheduler must re-determine if it can execute by its deadline because resources were not reserved during bidding. If so we are done. If not, the process continues (including FA, bidding, and preempting in some order based on policy or state information), or give up and run an error recovery routine for this task. Giving up also occurs if it becomes clear that a task being reallocated will not make its deadline, e.g., if deadline minus computation time is greater than the current time.
- The local component of the scheduler is able to decide if a set of tasks can meet their deadlines. That is, the guarantee routine of the scheduler is called to analyze whether a set of active tasks (usually a set of previously active tasks plus some new arrivals) at a local site can make their deadlines.
- An *important* issue is the effect of the reallocation algorithm on the deadlines of other tasks at this site. If we assume that the reallocation task runs in the system processor, then application tasks already guaranteed are not affected. However, new arrivals are not being looked at immediately because the system processor is now running the reallocation algorithm. If there were only one processor then we propose that when the reallocation algorithm is activated it must have a worst case time and it uses this to determine what tasks at this site will no longer make their deadlines because of this high priority reallocation task that has taken over. All such tasks are to be reallocated also. This process is not trivial when priority must be accounted for. The simulation results directly address this issue.
- Consider that when a host fails that it takes W seconds for it to be noticed, where W is very small. Each host begins the reallocation procedure as soon as is feasible (the reallocation procedure has a very high priority and preempts almost any task immediately, but in some cases it is possible for a higher priority task to continue to run and make the reallocation procedure wait). Consequently, it would be difficult to use a variation of this approach that requires synchronization of the decentralized reallocation tasks themselves. A solution is to have the reallocation procedure first identify what tasks at this site will now no longer make their deadlines because this reallocation task is executing, if any, and combine these tasks with the tasks that it is primary buddy for. At this point it may be best for each site to broadcast its state (maybe using an efficient form of TDMA to avoid collisions) and then each site uses this information to make one quick decision, estimating what the other sites will do to minimize instabilities. Actually, there are many variations of this idea which are possible including solutions which look centralized, but none guarantee that all tasks will be reallocated successfully.

In summary, the above discussion makes the following main points:

1. it illustrates where such a decentralized reallocation algorithm relies on a decentralized system-level scheduler.

2. In order to reallocate, the set of active tasks at a failed site must be known. In this approach, this information is known in a decentralized fashion. No single site has all the information. The reallocation decision itself for a given task is made at one site with backup, but possibly relying on cooperation with a decentralized scheduling algorithm. This approach makes decentralized reallocation quite reliable and efficient (if we avoid instabilities).

3. Focussed addressing, bidding, preempting locally and globally are all interrelated and the relationships among them are policies to be chosen by the user and/or is based on the current network state.

4. The algorithm works in parallel, partitioning the work. It is decentralized in most ways, uses best effort, and separates policy from mechanism. A given site makes a reallocation decision without consulting the other sites about their reallocation decisions. We did this for efficiency, but it is easy to envision an alternative algorithm where all the sites with tasks to reallocate would negotiate among themselves to come to some final decision about the location of the tasks. It is our feeling that more than 1-2 rounds of messages to accomplish such negotiation is too costly and this is supported by the simulation results.

5. A Final Note: The decentralized scheduler can be asked to perform a pseudo load balancing on the buddy tasks dynamically in order to keep the system ready for a failure, i.e., to increase the probability that all reallocation could be done locally. This seems too costly and is not implemented at this time.

Next we summarize the actual decentralized algorithm for aperiodic tasks by presenting it in pseudo code.

Decentralized Reallocation Algorithm

Tailored to Aperiodic Tasks

```
A. During Normal System Operation

1. On task guarantee

    • Choose a buddy using the random scheme

2. On task completion

    • deactivate buddies

B. On Host Failure
```

1. find the set of tasks that this host is responsible for reallocating - ones I am buddy for minus those that don't need reallocation because they are voting tasks or rtasks and meet minimum replication requirements even without reallocation [[this step is performed for efficiency]]

2. attempt a local guarantee by highest priority first {with or without preemption depending on a policy decision, but we choose without preemption as the default}

3. determine the new set S of tasks which must still be reallocated, call this {S}.

```
FOR EACH TASK IN {S}
  if laxity is small then
     preempt locally
     if task still cannot be guaranteed here and Enough_Time then
         perform FA'    *FA' means that some sight is always chosen;
                        *FA means that the best site above a threshold
                        * is chosen;
  if laxity is not small then
     if this site has the view that the network is busy then
        preempt locally
        if no solution yet then
           preempt globally
           if no solution then
              bidding
           else mark alternatives as exhausted
     else apply initial reallocation policy
          *an ordered vector [a,b,c,d] that describes order to
          *apply FA, Bidding,Local and Global Preemption
     [ 1. FA - net result is to send task to host X or try next
               alternative; calls Enough_Time
       2. Bidding - (n pairwise negotiations) - net result is to
               send task to X, identify next best site, or if
               if no good bid then try next alternative;
               calls Enough_Time; continue with next task while
               waiting for a reply;
       3. Preempt Locally - net result is success or try next
               alternative
       4. Preempt Globally - send task to X or try next
               alternative; calls Enough_Time;

       if we fail on last alternative then
               mark alternatives as exhausted ]

  if all alternatives are exhausted then
```

```
            consider task not reallocatable and perform error routine
               * error routine could be to ignore the task, ask user
               * to resubmit with a later deadline, etc.

   END_FOR;

 subroutine Enough-Time;
      * always called before a task is actually transmitted


      if (D-C-Transmit_Time-Processing_Time > Current_Time) then
           task is still viable and allow it to be sent
      else
         consider the task as failed to meet its deadline;
 end Enough_Time;
```

C. When Task Being Reallocated Arrives At X

```
 *such as task is marked as being reallocated, by what means it is
 *arriving (FA, Bidding, or Preempt Globally) and if via Bidding
 *whether this is the first or second choice.

 attempt to locally guarantee
 if successful then done
 else apply subsequent reallocation policy in some order [a,b,c,d]

       a. FA - transfer again or try next alternative; before
               transferring call Enough_Time
       b. Bidding - if Enough_Time then transfer to second best site
               else if already second best site then reject
       c. Preempt Locally - Successful or try next alternative
       d. Preempt Globally - if location found and Enough_Time
                then transfer else try next alternative

       if all alternatives are exhausted then
            consider the task as not reallocatable and perform
            the error routine
```

Note: If group bidding is utilized, then the above algorithm has to be modified slightly. This entails sending all the tasks through each step before proceeding to the next step. In the above algorithm and after a local guarantee for all tasks is attempted, the remaining highest priority task proceeds through all the steps, before we switch our attention to the next task.

However, even under the above strategy, some of the steps could be done in parallel, e.g., if we go out for a bid on a task, then while waiting we would proceed with the next task, or we could perform FA and bidding in parallel, as we did in our other decentralized scheduling work [8][10].

In this reallocation algorithm we must consider the tradeoffs between making a decision quickly using the current information a site has about the state of the network versus accepting additional delay and cost of obtaining new more up-to-date information. This decision must be made with respect to the facts that (a) tasks being reallocated have deadlines, and (b) the reallocation routine itself must be fast because it has a deadline and its execution time will affect the ability for tasks being reallocated to make their deadline. Our strategy is to make the decisions quickly for tasks with close deadlines, and *possibly* trade off some delay for improved information when a task has a longer laxity. In Section 6 we revisit the reallocation algorithm with respect to the principles of decentralized control, reliability and performance to further discuss these important issues.

Other alternative algorithms would attempt more exchange of information. This ranges from each site telling each other site about its state and its set of tasks to be reallocated, and then one decision is made, to multiple rounds of partial messages, and multiple decisions. The type, amount and frequency of data exchanged all could be a function of the load and/or deadlines of the tasks under consideration for reallocation. This would get quite complicated and it is not clear that it would perform well. Our philosophy is to begin with the above outlined strategy, refine it, and only consider additional information exchange when it is shown to clearly be worth it. Our simulation results indicate that it will rarely be worth it.

The above algorithm pseudo code could also apply to periodic tasks with several minor adaptations. On the other hand, with a few additional modifications, we can exploit the a priori knowledge of periodic tasks to improve overall performance. We, therefore, believe that it is necessary to treat these two types of tasks (periodic and a periodic) differently.

For the periodic tasks the changes would be made as follows:

- when the laxity is small, preempt locally as was done originally. However, if the task still cannot be guaranteed locally then forfeit the number of instances of this periodic task that we estimate is necessary before we can find a good home for this task. This estimate is F if there is a focussed host, otherwise B if we need to go out to bid. Both F and B are estimated based on past performance of focussed addressing and bidding, respectively.

- for each of the alternatives of FA, bidding, peempting locally and preempting globally, there is a possibility of attempting to reallocate periodic tasks so that no instances miss their deadlines (especially if laxity is long), but there is also the opportunity to cancel one or more upcoming instances of the periodic task with the intent of making all the deadlines after a time T has passed.

4 Evaluation

4.1 Description of the Simulation Programs

The main idea behind the simulation programs is to set up the state of the network just prior to the failure (called configuring the network), cause the failure, and then run the decentralized (centralized) reallocation algorithm on each non-failed host from the point of the failure onwards to some future time. In the simulation, we assume that each host has one application processor, one smaller system processor, and that there is only one host failure. The size of the network is an input parameter, and the results presented here are for sizes 4 and 8. The subnet is modeled as a bus with potentailly different delays for RFB's, BIDS, and task movement.

Configuring the network means that we a) generate a load for each host including the host that will fail, b) assign buddies for each generated task in the system, and c) for each host, generate a view of the loads at the other hosts of the network (to be used in Focussed Addressing). Consider each of these aspects of configuring the network in more depth.

a) There is no good single measure for non-periodic load in a real-time system because each task has a different deadline. We consider non-periodic load in the following way. First, we have to generate a load. This is done as follows: the computation time of a task is drawn from a uniform distribution between 15-50 units of time. The deadline of a task is drawn from another uniform distribution between 50-250 units of time where this value is added to the computation time. This means that the minimum deadline is 65 units and the maximum is 300 units. A collection of tasks is assigned to a host so that all of them make their deadlines and the SUM(C/200) = %LOAD, where %LOAD is a parameter of the test and 200 is an arbitrary but reasonable time interval. The problem is that %LOAD by itself is not a good indicator of load because it does not indicate the number of tasks needed to create a given load, nor the effect of deadlines. Consequently, we also use number of tasks and %LOAD(D) as two additional indicators of load. %LOAD(D) is a load factor that accounts for deadlines. It is given by

```
%LOAD(D) = 100% * [[C(1)/D(1) + (C(1)+C(2)/D(2)) + ...
                                   (C(1)+C(2)+...C(N)/D(N))]/N]
```

%LOAD(D) is a measure of the average laxity of tasks at a host.

An example will better clarify the meaning of the 3 load indicators. Let there be two hosts, one with 5 tasks and another with 6 tasks, all of which are guaranteed. The table below lists the task numbers, their deadlines, and the three load indicators (since no one indicator gives a true picture of the load). This example is taken from an actual simulation run:

TABLE 1: Example of Load Indicators

```
        HOST A                                   HOST B

  TASK     C     D   (76%)               TASK     C      D   (89%)

    1     23    67                         1     28     70
    2     72    85                         2     56     72
    3    106   112                         3     80     90
    4    140   174                         4     96    195
    5    157   181                         5    128    198
                                           6    163    219

%LOAD = 157/200 = 76%               %LOAD = 178/200 = 89%
No. of Tasks = 5                    Number of Tasks = 6
%LOAD(D) =   76%                       %LOAD(D) =  66%
All tasks make their deadlines.
```

On host A the %LOAD and %LOAD(D) metrics turned out to be identical, but this is rare as you will see from the data of the simulation runs. More typical is host B, where there is considerable difference between the two metrics.

For statistical validity, we changed the seeds of the simulation program random number generators. However, since the load factors are so dependent on the specific tasks generated and since the load is generated as part of the simulation, each new seed resulted in fairly high variations of %LOAD and %LOAD(D). However, the resultant success ratio was always in favor of the decentralized algorithm. Our approach was to make a large number of runs (over 200) and show typical results.

Two main input parameters of the simulation were determined by looking at the requirements of a real command and control application currently under development. In this application there are to be 20-100 general purpose processors, and 4-8 tasks active per host. Given this basic information, we then studied a wide range of external arrival rates, computation time requirements, and deadline values.

b) The second aspect of configuring the network is assigning buddies. This is trivial and is done with a random number generator. We generate one buddy per task in these simulation tests.

c) A host's view of the network is determined by taking the actual %LOAD of each host, generated as described above, and perturbing it by some amount chosen from a uniform distribution in the interval +− VIEW_SIZE_PERTUR. about the actual %LOAD. Each host's view of each other host is determined in this manner.

In these simulations the entire decentralized algorithm is implemented with two exceptions. One, we do not consider periodic tasks at this time, and two, the global preemption was not implemented because it seems to be unnecessary.

Many overheads are accounted for in the simulation program including the cost of performing the local check to determine if a task can execute at the site of the buddy (includes the cost of the guarantee routine O(n^2), the cost of performing the focussed addressing, and the cost of bidding. The overheads associated with bidding are substantial and include:

- RFB processing,
- transport of RFB,
- wait for dispatcher at receiving site,
- process incoming RFB and create a bid,
- transmit the bid,
- wait for dispatcher at original site after (some) bids are returned or a timer fires,
- process bids, and
- reassign task.

While the reallocation algorithm is executing, current tasks are also executing in parallel, and new external arrivals may occur. The external arrival rates are parameters of the individual test.

A second simulation program was implemented for the centralized model. The same scheme was used to configure the network so that identical situations existed for both algorithms. The centralized algorithm is assumed to have up to date information about all the hosts of the network and performs the reallocation for all the tasks on the failed processor using the same guarantee algorithm used at each site in the decentralized algorithm. Since the guarantee algorithm is O(n^2), we modelled the cost of the centralized algorithm execution as O(n^2). This is the only overhead of the centralized algorithm.

4.2 Performance Results

This section contains the description of the main simulation results. The simulation of a decentralized reallocation algorithm in a hard real-time environment is quite complicated and many parameters are of interest. Due to space limitations we cannot present results with respect to each parameter. Instead, we discuss the results of the most important and interesting ones. We present these results in several categories: 1) Basic comparison of the decentralized and centralized algorithms, 2) modifying unit costs of the algorithms, 3) the effect on external arrivals, 4) the effect of the C distribution, 5) the effect the quality of state information has on focussed addressing, and 6) larger networks.

Basic Comparison

Figures 5, 6 and 7 show typical results for different load patterns. Figure 5's load pattern is that Host 1 is lightly loaded, and the others are approximately equally loaded. Figure 6's load pattern is a more arbitrary unbalanced situation with no lightly loaded host, and Figure

FIGURE 5: UNBALANCED - ONE VERY LIGHTLY LOADED HOST

		LOAD					SUCCESS RATIO			
							DECENTRALIZED			CENTRALIZED
		H1	H2	H3	H4		H1	H2	H3	ALL HOSTS
	%L	.24	.51	.60	.60	RES.	1	3	0	
A)	%L (D)	.55	.38	.52	.54	LG	1	3	0	
	NO.	1	3	4	4	FA	0	0	0	
						BID	0	0	0	
						TOTAL		100%		75%
	%L	.24	.80	.78	.70	RES.	1	0	3	
B)	%L (D)	.55	.50	.77	.52	LG	1	0	3	
	NO.	1	5	4	4	FA	0	0	0	
						BID	0	0	0	
						TOTAL		100%		75%
	%L	.24	.82	.87	.88	RES.	2	2	2	
C)	%L (D)	.55	.53	.81	.67	LG	2	1	0	
	NO.	1	5	5	6	FA	3	0	0	
						BID	0	0	0	
						TOTAL		100%		33%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 6: UNBALANCED: NO LIGHTLY LOADED HOST

								SUCCESS RATIO		
			LOAD				DECENTRALIZED			CENTRALIZED
		H1	H2	H3	H4		H1	H2	H3	ALL HOSTS
	%L	.49	.64	.87	.88	RES.	2	2	2	
A)	%L (D)	.75	.44	.81	.67	LG	2	2	0	
	NO.	2	4	5	6	FA	0	0	0	
						BID	1	0	0	
						TOTAL		83%		33%
	%L	.49	.76	.89	.82	RES.	2	0	4	
B)	%L (D)	.75	.50	.80	.66	LG	1	0	2	
	NO.	2	4	6	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		50%		16.6%
	%L	.49	.64	.87	.88	RES.	2	2	2	
C)	%L (D)	.75	.44	.81	.67	LG	2	2	0	
	NO	2	4	5	6	FA	0	0	0	
						BID	1	0	0	
						TOTAL		83%		33%
	%L	68	.67	.89	.82	RES.	2	0	4	
D)	%L (D)	.72	.46	.80	.66	LG	1	0	2	
	NO.	3	4	6	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		50%		16.6%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 7: BALANCED

							SUCCESS RATIO			
			LOAD				DECENTRALIZED			CENTRALIZED
		H1	H2	H3	H4		H1	H2	H3	ALL HOSTS
A)	%L	.49	.51	.58	.59	RES.	2	1	1	
	%L (D)	.75	.37	.61	.51	LG	2	1	1	
	NO.	2	3	4	4	FA	0	0	0	
						BID	0	0	0	
						TOTAL		100%		75%
B)	%L	.68	.67	.70	.70	RES.	1	3	1	
	%L (D)	.72	.46	.74	.41	LG	0	3	1	
	NO.	3	4	4	5	FA	0	0	0	
						BID	0	0	0	
						TOTAL		80%		80%
C)	%L	.90	.83	.86	.88	RES.	1	3	2	
	%L (D)	.76	.54	.55	.88	LG	0	2	2	
	NO.	4	5	6	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		66%		20%
D)	%L	.90	.95	.89	.88	RES.	1	3	2	
	%L (D)	.76	.59	.59	.67	LG	1	2	1	
	NO.	4	6	5	6	FA	0	0	0	
						BID	0	0	0	
						TOTAL		66%		20%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

7's load pattern is that all hosts are approximately equally loaded. Within each load pattern, a range of loads is tested, and labelled A), B), C), etc. In these figures, the %LOAD, %LOAD(D) and number of tasks metrics are all presented for hosts 1-4 inclusive. Host 4 is the host that fails. These figures also show the success ratio of the decentralized versus the centralized algorithms. For the decentralized algorithm the figures indicate the number of tasks that each host are responsible for (RES), the number of tasks locally guaranteed (LG), the number of tasks guaranteed via focussed addressing (FA), and the number of tasks guaranteed via bidding (BID). Some later figures also show the number of tasks locally preempted (LPE) whenever this situation occurs. The success ratio in % of tasks from the failed host that are subsequently guaranteed by all means is presented under TOTAL, for both the centralized and decentralized algorithms. Note, that without a reallocation algorithm all tasks on the failed host will miss their deadline so the success ratio is zero. A major advantage of our approach is that it does not affect currently guaranteed tasks, so running the centralized or decentralized reallocation algorithm is a net improvement as long as at least 1 task is successfully reallocated, and this task does not cause one or more future external arrivals to not be guaranteed (see subsection below on external arrivals). Consider each of the Figures 5, 6 and 7, in turn.

Figure 5 shows that for specific loads A) and B) all tasks are guaranteed locally (100%) for the decentralized algorithm, while the centralized algorithm only guarantees 75% of the tasks. This type of result was typical for over a hundred runs made with non severe conditions. That is, it was typical for the decentralized algorithm (which nicely partitions the responsibility for tasks) to guarantee all, or nearly all, of the tasks while the centralized algorithm did not. Note that there is always the possibility that an individual task cannot be guaranteed because it had a very close deadline when the failure occurred. Under more demanding conditions, such as in load C) in Figure 5, Hosts 2 and 3 are not able to perform local guarantees for all the tasks (for example, Host 2 is responsible for 2 tasks and locally guarantees only 1), but subsequently make use of FA (since Host 1 is lightly loaded) to guarantee its second task. In total, 3 tasks are guaranteed via FA, so again 100% of the tasks are guaranteed. Note that bidding is not needed, so its overhead is not incurred. For load C) the centralized algorithm only guarantees 33% of the tasks, even though the system is capable of guaranteeing 100% of the tasks. This is strictly due to the computation time needed by the centralized algorithm to make its decision since its work is not partitioned. We made several additional runs (not shown in the Figure) where the centralized cost was modelled as linear. We did this to determine the effect on the results if the average cost of running the centralized algorithm was linear and not $O(n^2)$. For the same loads as shown in Figure 5, the success ratio for the centralized algorithm was 75%, 75% and 83% for loads A), B) and C) respectively. This is better, but still not as good as the decentralized case which includes a squared cost at each site (but the work is partitioned), and all the costs of FA, bidding, etc.

Figure 6 again shows the utility of partitioning the reallocation responsibility because a large percentage of the tasks are locally guaranteed under a different load pattern. For this load pattern, FA does not help because there is no host viewed as being lightly loaded, so no tasks are assigned via FA. Bidding is beneficial under loads A) and C) where an additional task is guaranteed, even with the high cost of bidding. It is typical, under this load pattern to find bidding useful about 50% of the time, while FA is not useful. From Figure 6, under all loads in this load pattern, the decentralized algorithm again clearly outperforms the centralized algorithm, i.e., the success ratio was 83% to 33%, 50% to 16.6%, 83% to 33%, and 50% to 16.6% for loads A), B), C) and D), respectively, in favor of the decentralized algorithm. When linear costs were assumed for the centralized algorithm, the success ratio improved to 66% for each of the four loads A), B), C) and D). This is better than the decentralized algorithm for cases B) and D) where the success ratio was only 50%. This anomaly is due to the fact of random assignment of responsibility in the decentralized algorithm where Host 3 was responsible for 4 tasks at squared cost. For a fairer comparison, we need to model a linear cost for the decentralized algorithm too. After doing this, the success ratio for the decentralized algorithm with linear cost improved to 100%, 83%, 83% and 83% for the four loads A), B), C), and D), respectively. Again, the decentralized algorithm is better than the centralized algorithm for all loads. The loads B) and D) show that our random assignment of buddies sometimes causes and imbalance of responsibility and a possible loss in performance. If this fairly rare condition is deemed important to avoid, it is easy to enforce a balanced level of responsibility. However, a balanced level of responsibility is not without cost, because a certain amount of coordination costs would be required to enforce a balance at all times. Our view is that this extra coordination cost is not worth it, because in all but a few rare cases we get excellent results with the random assignment of responsibility at zero coordination costs.

The results presented in Figure 7 indicate that when the network is balanced, neither FA nor bidding improve the success ratio. This is due to the fact that if a host cannot locally guarantee a task, other hosts are just as busy, and they might have even increased their load by their own local guarantees. Of course, in our many runs we did see isolated situations where FA and bidding increased the success ratio even in a balanced system, but such results are rare. The resultsof these tests indicate that a more adaptive version of our algrithm might attempt to recognize a balanced network load, ignore application of FA and bidding under these conditions, and instead, go directly to local preemption. Comparison of the success ratios of the decentralized and centralized algorithms (Figure 7) again show the superiority of the decentralized algorithm under this load pattern, i.e., 100% to 75%, 80% to 80%, 66% to 20%, and 66% to 20% in favor of the decentralized algorithm in spite of the fact that FA and bidding are not helpful. Hence the improvement is due to lower cost of the decentralized algorithm because it partitions the work.

Unit Costs

In all the previously presented results, the decentralized algorithm outperforms the centralized algorithm. Obviously, the comparison is heavily dependent on the costs assumed for each of the algorithms. We were realistic about these costs. For example, since the algorithms must attempt a guarantee for each task for which it is responsible, and the guarantee algorithm is $O(n^2)$ we used a squared cost for each algorithm. However, implicit in applying this cost is some unit cost which is then squared. In the previous runs the unit cost was 5 units. Figure 8 shows the results when we reduce this unit cost to 3 units and then to 1 unit. The decentralized algorithm is still better than the centralized algorithm at all unit costs, althought the difference is smaller at smaller unit costs. The results shown in Figure 8 span the three load patterns presented in Figures 5-7 inclusive. For example, load A) in Figure 8 is a representative of the imbalanced load pattern, and the results are 66.6% to 50%, 50% to 50% and 50% to 16.6% for unit costs 1, 3, 5, respectively, in favor of the decentralized algorithm. Similar results are shown in Figure 8 for the other two load patterns, i.e., load B) is the balanced load pattern, and load C) is the "1 host lightly loaded, the others approximately equally loaded" load pattern.

External Arrivals

Reallocation adds extra work to each site. If this extra work causes subsequent external arrivals to miss their deadlines, then the net effect of the reallocation is diminished or possibly even negative. Two of the more important parameters in determining the effect of reallocation on external arrivals, is the load on the network at the time of the failure, and the external arrival rate. We considered a range of interarrival times for tasks arriving from the external world (a poisson process) with average times 90, 70, 60, 50, 40 and 30 time units at each host. Since the average service time is 38 units, the runs with 40 and 30 time units as averages are very demanding, and overloaded, respectively. For network loads similar to the ones found in earlier portions of this paper, the effect of reallocation on external arrivals was minimal (almost 0) for average interarrival times of 50, 60, 70, and 90 time units. This is primarily due to the fact that the extra load imposed by reallocation is not considerable compared to the laxity of a new external arrival. In other words, the new arrivals tend to let the reallocated tasks run first, and there still exists enough time for them to execute before their deadlines. This condition does not hold when external arrivals are very frequent. As a typical example consider the interarrival rate of 40 units. A typical test result was that the decentralized algorithm success ratio was 5 out of 8 tasks, but subsequently 2 external arrivals were lost - a net gain of only 3 tasks, not 5. For the same test, the centralized algorithm successfully reallocated 4 out of 8, but lost 5 external arrivals for a net loss of 1 task. When the interarrival time was decreased to 30 units, the net effect for the decentralized algorithm was negative (5 reallocated and 6 lost external arrivals), and neutral for the centralized algorithm (4 reallocated and 4 lost external arrivals). Tests results such as these substantiate an intuitive notion, that if the system is overloaded, there is no net worth in performing reallocations, and, in fact, it can be harmful if the algorithm doesn't have mechanisms to turn itself off. Consequently, we recommend adding a factor to our algorithm that would attempt to monitor the load on the system, and turn off reallocation when the load is too high. This simple scheme would have to be slightly modified if it was policy that the priority of tasks was the prime factor during reallocation.

Computation Time Distribution

The laxity (deadline minus remaining computation time) of a task is an extremely impor-

FIGURE 8: UNIT COSTS

		LOAD				UNIT COSTS	SUCCESS RATIO	
		H1	H2	H3	H4		DECENT.	CENTR.
A)	%L	.49	.76	.89	.82	1	66.6%	50%
	%L(D)	.75	.50	.80	.66	3	50%	50%
	NO.	2	5	6	6	5	50%	16.6%
B)	%L	.68	.79	.78	.74	1	60%	40%
	%L(D)	.72	.52	.60	.66	3	60%	40%
	NO.	3	5	5	5	5	40%	20%
C)	%L	.24	.68	.61	.63	1	100%	75%
	%L(D)	.55	.46	.69	.54	3	100%	100%
	NO.	1	4	3	4	5	100%	75%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

tant factor in the success of any on-line algorithm. If all laxities are very small, then there is no point to on-line decision making; as laxities increase we can expect better and better results from on-line algorithms. Consequently, if we run tests with increasing the computation time requirements with fixed deadlines (thereby decreasing laxity), then we expect a net decrease in the number of tasks successfully reallocated. This result was observed over many tests. However, another issue is the relative performance of the centralized to the decentralized algorithms over a range of computation times. Recall that most of the results presented earlier are with a computation time for a task given by a uniform distribution in the range of 15-50 units. In adition, we tested under uniform distributions over the ranges 30-60, 40-70, 10-90, and 30-100 units. The decentralized algorithm performed better than the centralized algorithm for all three load patterns and over all computation time distributions tested. To save space we will not present the actual data from which this conclusion was reached. Of course, particular tasks with high computation time requirements were not often successfully reallocated in either case, as is to be expected.

Quality of State Information

Focussed addressing is one aspect of our reallocation algorithm which requires approximate information about the state of other hosts. In these tests we used an approximation of the %LOAD factor as an indicator of load. We then tested the effect on performance of an error in a host's view of this %LOAD factor. All previous tests used a +- 10% factor, meaning that a host j's view of another host i was off by +-10%. We then tested +-20%, +-30%, and +-40% for each of the three load patterns.

Figure 9 shows that when the load pattern was all hosts equally busy and the load was approximately 75% at each site, the success ratio of 40% was unaffected by poor state information. This is, in part, due to the fact that in most cases a host's view of another host was that this other host was too busy for it to be considered as a FA host, and, in part, to the fact that when a task was sent to a perceived FA host it was not accepted due to the heavy load at that site.

Figure 10 illustrates an *improvement* in success ratio from 33% to 50% when there is poor information. This is typical under this load pattern because under heavy but unbalanced loads, a site often erroneously considers a site as lightly loaded, sends work, and luckily, there is sometimes enough free time to guarantee it. The FA decision is made quickly at low cost. Bidding, because of its higher cost, is not able to guarantee these extra tasks in this case (but it can in other cases - see next paragraph). We have found similar results in our other decentralized scheduling work [8][10], and the solution is to perform FA on the host considered most lightly loaded, but in parallel proceed with bidding. Employing this scheme we were able to attain better overall results in normal task scheduling [8][10] and would expect the same results with respect to reallocation.

Figure 11 is for the load pattern where one site is lightly loaded and the others are heavily loaded. Here FA is used extensively (i.e., 50% of the tasks successfully reallocated were via FA) when the quality of the information is good (as in the +-10% case). This data appears in the section labelled 10) in Figure 11. As the quality of information degrades, fewer tasks

FIGURE 9: FOCUSSED ADDRESSING - LOAD A

							SUCCESS RATIO	
		LOAD				%PERTURB.	%SR	%FA
		H1	H2	H3	H4			
	%L	.68	.79	.78	.74	+-10	40%	0%
A)	%L (D)	.72	.52	.60	.66	+-20	40%	0%
	NO.	3	5	6	6	+-30	40%	0%
						+-40	40%	0%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 10: FOCUSSED ADDRESSING - LOAD B

							SUCCESS RATIO	
		LOAD				%PERTURB	%SR	%FA
		H1	H2	H3	H4			
	%L	.68	.67	.89	.82	+-10	33%	0%
B)	%L (D)	.72	.46	.80	.66	+-20	33%	0%
	NO.	3	4	6	6	+-30	50%	33%
						+-40	50%	33%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

FIGURE 11: FOCUSSED ADDRESSING - LOAD C

								SUCCESS RATIO	
			LOAD					DECENTRALIZED	
		H1	H2	H3	H4		H1	H2	H3
	%L	.24	.82	.87	.88	RES.	2	2	2
10)	%L(D)	.55	.53	.81	.67	LG	2	1	0
	NO.	1	5	5	6	FA	3	0	0
						BID	0	0	0
						TOTAL		100%	
	%L					RES.	2	2	2
20)	%L(D)		SAME			LG	0	0	0
	NO.					FA	2	0	0
						BID	2	0	0
						TOTAL		66.6%	
	%L					RES.	2	2	2
30)	%L(D)		SAME			LG	2	0	0
	NO.					FA	1	0	0
						BID	2	0	0
						LPE	0	1	0
						TOTAL		100%	
	%L					RES.	2	2	2
40)	%L(D)		SAME			LG	2	0	0
	NO.					FA	0	0	1
						BID	3	0	0
						TOTAL		100%	

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

are moved by FA (from 3 tasks, to 2 tasks, to 2 tasks, then to only 1 task as the view degrades from 10-20-30-40%), but the slack is taken up by bidding which increases from 0 tasks, to 2 tasks, to 2 tasks, to 3 tasks as the view degrades from 10-20-30-40%. It is for this reason that we cannot just use FA, but need to proceed with FA and bidding in parallel. In this test the deadlines of the tasks to be reallocated were sufficiently long to allow bidding to be successful. It is easy to hypothesize other situations where the poor quality of state information would reduce FA, as in this figure, but where subsequent bidding would then not be successful because deadlines are too close. This is the situation which would be most affected by poor quality of state information. In summary, we believe that our algorithm is quite robust to state information sharing since it is used in limited contexts, and the affect of its degradation is usually ameliorated by bidding.

Larger Networks

Having performed the majority of our tests simulating a four host network, we increased the size of the network to 8, where host 8 fails. Over many tests we observed the same general result, the decentralized algorithm significantly outperforms the centralized algorithm. Figure 12 shows typical test results. For load A), 66% of the tasks are reallocated via local guarantee (and none by FA, bidding or local preemption), while the centralized algorithm is successful with only 16.6%. In load B) our decentralized algorithm is successful with 100% of the tasks, but does preempt a low priority task at host 1. This compares to only a 50% success ratio for the centralized algorithm under the same conditions. Load C) produces as 100% success ratio for the decentralized algorithm, primarily by FA, while the centralized algorithm only guarantees 33%. Load D) is even more dramatic, producing a 100% to 0% result in favor of the decentralized algorithm. However, in load D) the decentralized algorithm does preempt 3 low priority tasks to achieve this success ratio, one each at sites 2, 4 and 5. Local preemption occurred because tasks were sent to sites 2, 4 and 5 via FA, but were subsequently not guaranteed. At this point, bidding was not attempted because there was not enough time for bidding, so preemption was tried and was successful. If bidding is attempted before preemption and bidding is not successful, then chances are good that local preemption will also not be successful. That is, the algorithm will have waited too long before preempting. This is why we rarely see any local preemptions in these test results. It is a policy decision that the designers of the system under consideration must make - the order in which to employ the various parts of the reallocation algorithm.

Finally, the results from load E) are presented to emphasize that the local guarantee is quite useful, and that bidding is also sometimes useful. Load E) results in a 100% to 33% success ratio in favor of the decentralized algorithm. Note that while bidding is not always useful, it is useful in certain circumstances where otherwise there would be no chance at success. Since all the overheads are accounted for in bidding, using it helps sometimes, and does not cause negative results if carefully done.

FIGURE 12: LARGER NETWORK - 8 NODES

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.90	.95	.89	.88	.84	.86	.74	.91
A)	%L (D)	.76	.59	.72	.67	.75	.79	.53	.72
	NO.	4	6	5	6	5	6	5	6

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES.	2	1	0	1	1	0	1	
LG	2	1	0	1	0	0	0	
FA	0	0	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
TOTAL	66.6%							16.6%

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.24	.82	.87	.88	.83	.85	.85	.60
B)	%L (D)	.55	.53	.81	.67	.74	.63	.66	.72
	NO.	1	5	5	6	4	5	5	4

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES.	1	0	0	1	1	1	0	
LG	1	0	0	0	1	1	0	
FA	0	0	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
LPE	1	0	0	0	0	0	0	
TOTAL	100%							50%

FIGURE 12 CONTINUED:

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.49	.47	.72	.79	.78	.90	.79	.55
C)	%L (D)	.75	.35	.75	.54	.56	.70	.71	.65
	NO.	2	3	3	4	4	5	6	3

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES.	0	0	1	0	0	1	1	
LG	0	0	0	0	0	0	0	
FA	0	2	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
TOTAL				100%				33%

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.45	.48	.60	.55	.49	.52	.30	.57
D)	%L (D)	.76	.59	.78	.78	.71	.79	.72	.84
	NO.	4	6	7	7	6	6	4	7

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES.	2	2	0	0	1	0	2	
LG	1	1	0	0	1	0	1	
FA	0	0	0	0	0	0	0	
BID	0	0	0	0	0	0	0	
LPE	0	1	0	1	1	0	0	
TOTAL				100%				0%

FIGURE 12 CONTINUED:

LOAD

		H1	H2	H3	H4	H5	H6	H7	H8
	%L	.45	.48	.75	.79	.78	.90	.82	.87
E)	%L(D)	.31	.64	.73	.58	.57	.70	.51	.68
	NO.	3	3	5	6	6	6	5	6

SUCCESS RATIO

	DECENTRALIZED							CENTRALIZED
	H1	H2	H3	H4	H5	H6	H7	ALL HOSTS
RES.	0	1	2	0	2	0	1	
LG	0	1	2	0	1	0	1	
FA	0	0	0	0	0	0	0	
BID	1	0	0	0	0	0	0	
TOTAL				100%				33%

C DISTRIBUTION - 15-50 UNITS
D DISTRIBUTION - 5-200 UNITS
SQUARED COST (UNIT = 5)

5 Reallocation Algorithm Revisited

Since we are interested in decentralized control (and coordination), reliability and performance, let's revisit the six main steps of the algorithm with respect to these issues. See Table 2.

During normal system operation, steps 1 and 2 (Table 2), the algorithm is highly autonomous, reliable and performs well. There is no need for coordination, but in some sense there has been an a priori agreement (a protocol) made by each site. The same argument applies during host failure for steps 3, 4, 5, and 6a, b, c. Only in steps 6d and e does explicit cooperation occur, but at a high cost. Based on the simulation results, our decision to treat bidding and global preemption as last resorts seems to be a good one.

TABLE 2: LEVEL OF COOPERATION

Algorithm Step	Dec.Control	Rel.	Perf.
DURING NORMAL SYSTEM OPERATION			
1)choose buddies	each site on a per task basis; decisions made decentralized; no coordination;	all up sites can continue	overhead of choosing buddies is small and distributed; ovhd of broadcast and commit acceptable;
2)deactivate buddies	task completion at any site invokes the deactivate operation; decisions made decentralized; no coordination;	at task level; all up sites can continue	ovhd is small
ON HOST FAILURE			
3)find set resp. for	n sites, roughly in parallel determine this; no cooperation	each site can perform autonomously;all up sites can cont.	fast;no delay for cooperation
4)attempt local guarantee	same	same	same

5)for small laxity preempt locally	same	same	same
6)			
a) for not small lax., check state, apply policy	same	same	same
b) -FA	same	same	same
c) -Preempt Locally	same	same	same
d) -bidding	n-pairwise negot.	high	slow
e) -Preempt Globally	n-pairwise negot.	high	slow

The above table illustrates that any sophisticated *decentralised* algorithm will contain many steps, and/or phases being executed by multiple distributed agents. Each step of the algorithm at each agent of the decentralized algorithm could exhibit a different degree of decentralization. There is a basic tradeoff between information exchange used for coordination and/or negotiation among the decentralized agents running the algorithm and speed of execution (i.e., of making the decision). As in the above reallocation algorithm, it seems that most decentralized system algorithms will contain *points of autonomy* and *points of coordination*. For example, it is easy to modify the above algorithm by changing points of cooperation. Suppose we add a step 5.5 which required that every host exchange state information. This cooperation would occur after local guarantees, but before we attempt any distributed reallocation. This exchange would increase the quality of the state information available, but cause additional delays. The benefit of such added cooperation would have to be proven by a performance evaluation. If one eliminates coordination altogether, you end up with a totally decentralized algorithm where each agent is making decisions based only on local information; bound to cause confusion and poor performance. Finding the cost-performance tradeoff curve for a given algorithm in a given environment so that the algorithm has the "best" amount of coordination to achieve good performance under acceptable cost is the design problem generally faced. We have attempted to make these tradeoff choices in designing the algorithm presented above. Our conclusion is to always attempt a local guarantee first, then always perform FA. We need to then have a policy decision as whether to perform local preemption or bidding next.

Decentralized algorithms such as our scheduling [6][8][10] and reallocation algorithms contain many parameters and policy decisions. Extensive simulation has taught us that there is no one policy or set of parameter settings that work under all conditions. We have also been able to ascertain a number of ad hoc rules concerning scheduling and reallocation. We hypothesize that certain problems like stability and robustness may be controlled more effectively from a meta level controller than from within the scheduling algorithm itself. It is expected that in a complex application environment, policies will change both over time and due to external events, and parameter settings will need to be varied to reflect changing loads and requirements. In addition, these environments would like to retain control over the algorithm to some extent especially when dealing with life critical situations. It is also true that many systems deal with or are characterized by situations that occur at different rates. A multi-level controller can deal with such situations. Consequently, we hypothesize the need for a Meta Level Controller which will be heuristic, rule-based, and distributed. It will execute asynchronously and/or at a different rate from the lower level controllers, being invoked by special users and/or when conditions change considerably, and/or at critical points.

The Meta Level Controller should interface with human operators providing:

- a description of the policy used and why it is used,
- a description of the parameters used and to the best of its ability why they are used,
- commands to alter the policy,
- commands to alter the parameter settings (including the rates of change of the parameter settings),
- add or change reasoning rules, and
- maintaining and collecting the proper information to evaluate itself and suggest changes, or new rules (long range and may not be automatic).

6 Summary

We have developed and analyzed a decentralized task reallocation algorithm for hard real-time systems. The main properties of the algorithm are that it is decentralized and reliable, it specifically considers deadlines of tasks, it attempts to utilize all the nodes of a distributed system to achieve its objective, it is fast, it handles tasks in priority order, and it separates policy and mechanism. Another significant advantage of our approach is that it allows continued operation of the guaranteed tasks on the non-failed processors while reallocation is in progress. The reallocation algorithm itself is $O(n)$, but it invokes the guarantee algorithm which is $O(n^2)$. Consequently, the reallocation process is $O(n^2)$. An extensive performance analysis of the algorithm via simulation has shown that it is quite effective in performing reallocations, and is significantly better than a centralized approach. In these tests, the number of tasks to be reallocated varied from 3–8. In other environments where significantly larger number of tasks are to be reallocated, the decentralized algorithm should perform even better with respect to the centralized algorithm. On the other hand, when the number of tasks to reallocate is large, neither algorithm will be highly successful, therefore, only the n highest priority tasks should be considered for reallocation where n is probably in the range of 5–10.

While the algorithm presented handles both periodic and non-periodic taske, we have only tested non-periodic tasks. In most cases the difficulty in reallocating a periodic task will be with guaranteeing the current active instance of the periodic task (which can be thought of as a non-periodic task). Consequently, if one can guarantee the current active instance of a periodic task at a site, then it is highly likely that subsequent invocations of the periodic task can also be guaranteed there. Using this simple approximation as an argument, we can expect good performance in reallocating periodic tasks too. The ability to reallocate periodic tasks will increase as we take advantage of allowing the system to miss the next m invocations of a periodic task, as long as it is guaranteed from that point forward in time. The value m would be particular to each task.

References

[1] Carlow, G., Architecture of the Space Shuttle Primary Avionics Software System, *CACM*, Vol. 27, No. 9, Sept. 1984.

[2] Dasarathy, B., Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them, *IEEE Transaction on Software Engineering*, Vol. SE-11, No. 1, January 1985.

[3] Leinbaugh, D., Guaranteed Response Times in a Hard Real-Time Environment, *IEEE Trans on Soft Eng*, Vol. SE-6, January 1980.

[4] Liu, C, and J. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *JACM*, Vol. 20, No. 1, January 1973.

[5] Mok, A., and L. Dertouzos, Multiprocessor Scheduling in a Hard Real-Time Environment, *Proc. Seventh Texas Conf. Comp Sys*, Nov. 1978.

[6] Ramamritham, K. and J. Stankovic, Dynamic Task Scheduling in Distributed Hard Real-Time Systems, *IEEE Software*, Vol. 1, No. 3, July 1984.

[7] Ramamritham, K., J. Stankovic, and W. Zhao, Scheduling Tasks with Resource Requirements in Hard Real-Time Systems, to appear *IEEE Transactions on Software Engineering*.

[8] Ramamritham, K., J. Stankovic, and W. Zhao, Distributed Scheduling of Hard Real Time Tasks Under Resource Constraints in the Spring System, submitted to *IEEE Transactions on Computers*, Feb. 1986.

[9] Sha, Lui, J. Lehoczky, and R. Rajkumar, Solutions for Some Practical Problems in Prioritized Preemptive Scheduling, submitted to 1986 Real-Time Systems Symposium, 1986.

[10] Stankovic, J, K. Ramamritham, and S. Cheng, Evaluation of a Bidding Algorithm for Hard Real-Time Distributed Systems, *IEEE Transactions on Computers*, Vol. C-34, No. 12, Dec. 1985.

[11] Stankovic, J and D. Towsley, Dynamic Reallocation in a Highly Integrated Real-Time Distributed System, *Proc. Sixth Int. Conf. on Distributed Computing Systems*, May 1986.

[12] Ward, Paul, and S. Mellor, ***Structured Development for Real-Time Systems***, Yourdan Press, Vol. 1 and Vol. 2, N.Y., N.Y., 1985.

DESIGN OF EFFICIENT PARAMETER ESTIMATORS FOR DECENTRALIZED LOAD BALANCING POLICIES[1]

Spiridon Pulidas[2], Don Towsley[3], Jack Stankovic[4]
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 87-79
August 20, 1987

Abstract

In this paper we study the problem of efficiently determining the optimum parameter values present in a decentralized threshold load balancing policy. The purpose is to improve the performance of a distributed system (e.g. minimize the average response time of a job) by using the processing power of the entire system. This is done by transfering jobs from heavily loaded nodes to lightly loaded nodes. A distributed optimization algorithm, where each host computes its own threshold value on-line is adapted to the selected load balancing policy. The algorithm requires that each host computer be able to estimate the change in throughput and expected queue length d· : to a change in either the threshold or the job arrival rate. Two efficient estimation techniques are proposed for estimating the required gradient information. These techniques are imbedded in the distributed threshold updating algorithm and simulation results are obtained for its behavior in a static as well as in a changing system environment.

[1] This work was supported by RADC under contract RI-448896X.

[2] Department of Electrical and Computer Engineering.

[3] Department of Computer and Information Science.

[4] Department of Computer and Information Science.

Table of Contents

1. INTRODUCTION

We consider distributed computer systems consisting of multiple host computers interconnected by a communication network. Jobs arrive at each host computer according to some arrival process and can be processed either locally or at other hosts after being transfered through the communication network. The results of jobs processed remotely are returned to the origin host computer. Communication delays are incurred due to the transfer of remote jobs and their results. In such an environment, *load balancing* attempts to improve the performance of the distributed system (e.g. to minimise the mean response time of a job) by using the processing power of the entire system to smooth-out periods of high congestion at individual nodes. This is done by transfering jobs from heavily loaded nodes to lightly loaded nodes.

Throughout this work we consider a class of *threshold* load balancing policies, that have been shown to be useful in several studies [1,2,3,4,5,6,7]. Specifically, we adopt a threshold policy studied by Eager, Lasowska and Zahorjan [5]. In this policy, jobs at each host are divided into two classes; local and remote jobs. *Local jobs* are those processed at the site of origination and *remote jobs* are those transfered for processing to another node. A job arriving to a node from the external world is processed locally only if the current queue length is less than a *threshold* parameter. Otherwise, the job is sent for remote processing to another host selected according to some probability distribution. Remote jobs are always accepted by the destination hosts and therefore jobs can move at most once. Both local and remote jobs are processed according to a first-come-first-served (FCFS) discipline at a given host.

Obviously, such a threshold policy has control parameters (e.g. threshold values and transfer probabilities for every host computer), that require fine-tuning in order to yield optimal or near-optimal performance. An efficient distributed algorithm for determining the optimum values for parameters present in the decentralised threshold scheduling policy has been proposed by Lee and Towsley [1]. The algorithm is iterative in nature and at each iteration the load balancing parameters at each host are updated. This algorithm requires that each host computer be able to estimate the change in throughput and expected queue length due to a change in either the threshold or the job arrival rate. The host computers exchange this gradient information with each other and use these quantities to update the load balancing parameters for the next iteration of the algorithm.

Our approach towards estimating the gradients with respect to the threshold relies on the assumption that the arrival process is exponentially distributed. By adding a small perturbation to the observed parameter (i.e. a small decrease in the threshold), the estimator we have developed determines the effect of the change on the performance metric of interest (i.e. throughput, expected queue length), taking advantage of the *memoryless property* of the arrival process distribution. The estimator is originally presented for a system with Poisson arrivals and then is modified, so that it can be applied to a system with general distribution of arrival times but exponentially distributed service time. The arrival rate is estimated during the execution of the algorithm. The memory requirements necessary for the implementation of the estimation algorithm (i.e. number of counters required) are very low. The major advantage of this technique is its *on-line* potential, since it effectively attempts to provide performance sensitivity information while the actual system is running. The proposed estimation technique has been evaluated through simulations and the results were very accurate when compared to the analytically obtained results.

A different estimation method is proposed to obtain estimates with respect to the arrival rate. The method is based on a class of theorems derived from Likelihood Ratios and is extremely well suited to regenerative systems. A busy cycle of the processor has been used as the regeneration period. We have used the estimator in simulations with a very low increase in running time or memory requirements.

In Section 2, we describe the system model and the Load Balancing policy in detail. We consider the problem of determining optimum values for the parameters present in the load balancing policy and we adapt the distributed optimisation algorithm developed by Lee and Towsley [1]. The necessary gradient information required by the above algorithm is also identified in this section. In Section 3, we present *Estimator-A*, which provides estimates of gradients with respect to the threshold. Simulation results from the application of the proposed estimator on various kinds of systems are also presented. The main example, considered in this section, is the application of Estimator-A on a processor modeled as a single-server queueing system with two classes of jobs (local and remote jobs), where only one of these classes is governed by the threshold policy. In Section 4, *Estimator-B* is introduced in order to provide gradient information with respect to the arrival rate. The accuracy of the estimated gradients, as well as the convergence properties of the proposed estimator are evaluated through simulations. In Section 5, both Estimators A and B are imbedded in the decentralized threshold scheduling policy discussed in Section 2. Simulation results are presented for a system with five host computers modeled as single server queueing systems. It turns out that after a finite number of algorithm iterations, the behavior of the system, in a static environment is confined in a neighborhood of the optimal performance. Furthermore, a great improvement in the performance measure (average response time of a job in the system) has been observed in the system executing the distributed load balancing algorithm, compared to a system with no load balancing at all. Finally, we summarize our work in Section 6.

2. A DECENTRALIZED LOAD BALANCING SCHEME

2.1. System Model

The system considered in this work consists of N autonomous host computers interconnected by a communication network (Figure 2.1). Jobs arrive at each host from the external world according to some arrival process with rate λ_i, $i = 1, 2, 3, ..., N$. Jobs originating at host i can be processed either locally or at any other host $j \neq i$. For the sake of simplicity, it is assumed that jobs processed at each host have the same service time distribution, regardless of the origin host. The results of a job transfered for remote service are returned to the origin host computer. Communication delays are incurred during both transfers. Each host has a communication server that takes care of the job transfers between computers.

2.2. Load Balancing Policy

Jobs at each host computer are divided into two classes; namely *local* and *remote* jobs. Local jobs are those that are processed at the site of origination and remote jobs are those that have been transfered from other hosts for remote processing. Let L_i denote the total number of jobs at host computer i. The following threshold load balancing policy is considered:

- If a job arriving at host i from the external world finds that $L_i < T_i$, where T_i is a threshold parameter associated with host i, it is processed locally.
- If an external job arriving at host i finds $L_i \geq T_i$, then this job is transfered for remote service to a host $j \neq i$ with probability P_{ij}, where $\sum_{j \neq i} P_{ij} = 1$.
- Remote jobs arriving at host i from other hosts are always accepted.
- Jobs arriving at each host are processed according to a first-come-first-served (FCFS) basis.

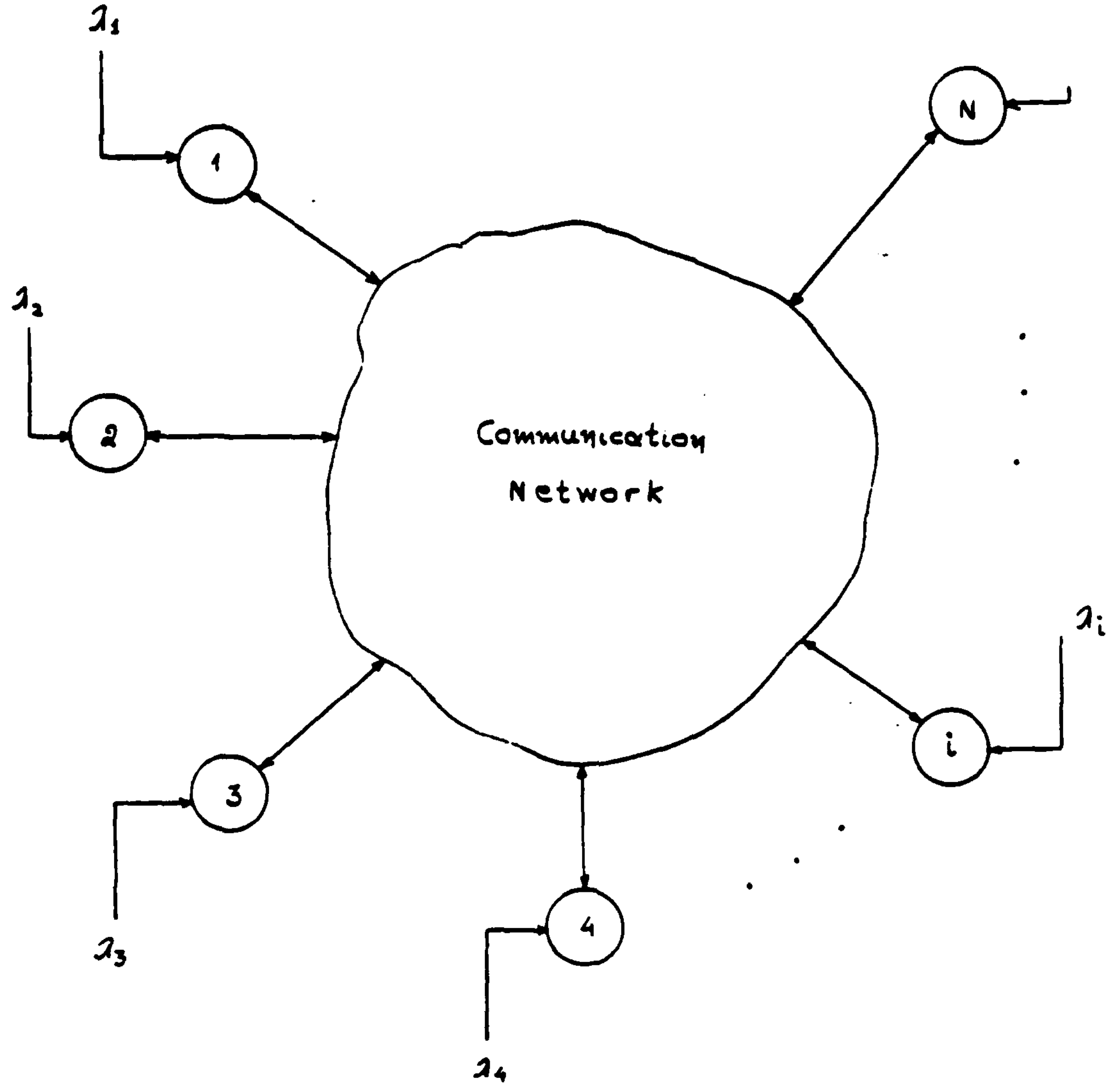


Fig. 2.1 - The System Model.

2.3. Optimal Load Balancing Problem

The threshold policy described in section 2.2 has control parameters (i.e. threshold value and transfer probabilities for each host computer) which require fine-tuning in a changing system environment. We select the mean response time of a job as the performance measure. Each host computer is modeled as a single-server queueing system with two classes of jobs (Figure 2.2). Let $\lambda_i^{(l)}$ and $\lambda_i^{(r)}$ be the throughput of local and remote jobs respectively at host i. Let also $f_i(\lambda_i^{(l)}, \lambda_i^{(r)})$ and $g(\Lambda^{(r)})$ denote the mean queue length at node i and the communication network respectively, where $\Lambda^{(r)} = \sum_{i=1}^{N} \lambda_i^{(r)}$. The mean response time $E[R]$ of a job in the system is given by the following formula [2]:

$$E[R] = \frac{\sum_{i=1}^{N} f(\lambda_i^{(l)}, \lambda_i^{(r)}) + g(\Lambda^{(r)})}{\Lambda} \qquad (2.1)$$

where $\Lambda = \sum_{i=1}^{N} \lambda_i$. Under the described threshold policy the optimization problem can be stated as follows [1]:

MINIMIZE $\quad \Lambda E[R]$

with respect to

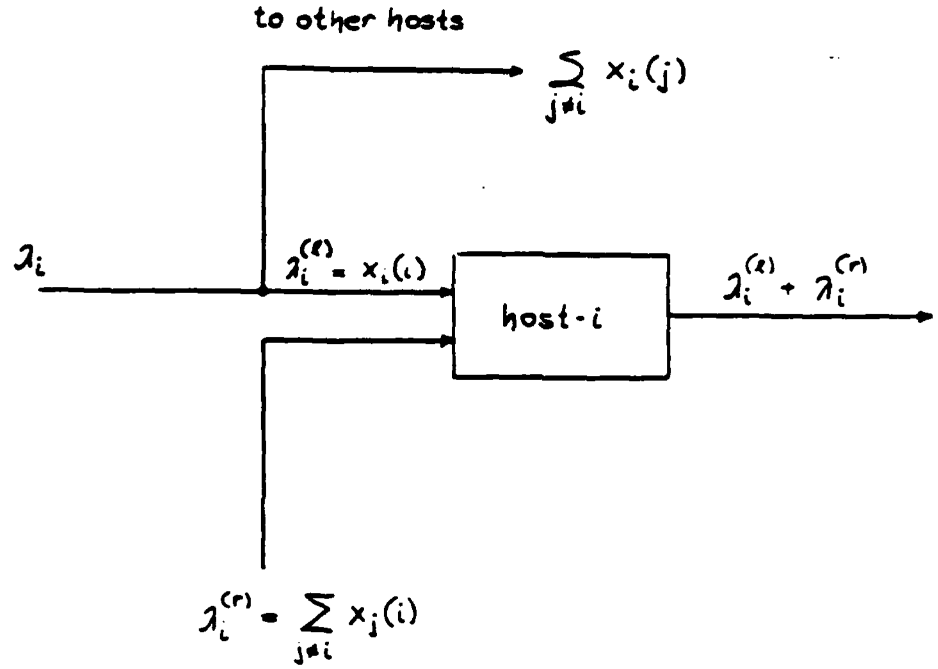
T_i's and P_{ij}'s,

under the constraints

T_i is non-negative integer, $\quad i = 1, 2, ..., N,$

$\sum_{j \neq i} P_{ij} = 1, \quad i = 1, 2, ..., N.$

This is an integer optimization problem with respect to integer and real variables. This kind of problem cannot be solved exactly and a heuristic algorithm is required for its solution. Such an approach is proposed in [1], where a distributed optimization algorithm is used.


to other hosts
$\sum_{j\neq i} x_i(j)$
λ_i
$\lambda_i^{(\ell)} = x_i(i)$
host-i
$\lambda_i^{(\ell)} + \lambda_i^{(r)}$
$\lambda_i^{(r)} = \sum_{j\neq i} x_j(i)$

<u>Fig. 2.2</u> - Host i as a single server queueing system with two classes of jobs.

By distributed algorithm we mean that each host computer performs a portion of the whole computation (it does not solve the entire optimization problem) and collaborates with each other to solve the problem by exchanging some information.

Let $x_i(j)$ denote the flow of jobs originating at host i and processed at host j. Obviously, $x_i(i) = \lambda_i^{(l)}$ and $\sum_{j \neq i} x_j(i) = \lambda_i^{(r)}$ (Figure 2.2). For given values of T_i's and P_{ij}'s, the steady state flows $x_i(j)$'s can be computed. However, it may not be possible to compute integer T_i's for arbitrary values of $x_i(j)$'s. We ignore this consideration and treat $x_i(j)$'s as non-negative real variables. The optimization problem can be reformulated as follows:

<u>MINIMIZE</u> $\quad \Lambda E[R]$

<u>with respect to</u>

$x_i(j)$'s,

<u>under the constraints</u>

$\sum_{j=1}^{N} x_i(j) = \lambda_i, \quad i = 1, 2, ..., N,$

$x_i(j) \geq 0, \quad i = 1, 2, ..., N.$

Necessary conditions for the optimal solution of the above problem can be derived from the Kuhn-Tucker conditions [8] as follows. For all j's,

$$\frac{df_j}{dx_i(j)} + (1 - \delta_{ij}) \frac{dg}{dx_i(j)} \begin{cases} = C_i, & \text{for } x_i(j) > 0 \\ \geq C_i, & \text{for } x_i(j) = 0 \end{cases} \tag{2.2}$$

where C_i is some constant (Langrange multiplier), and δ_{ij} denotes the Kronecker delta function. This is true for each host computer i, $i = 1, 2, ..., N$. Note that $(1 - \delta_{ij})$ has been introduced since local jobs do not experience communication delay. The derivative $df_j/dx_i(j)$ represents the incremental delay incurred for jobs processed at host j due to the job flow from host i to host j. The derivative $dg/dx_i(j)$ represents the incremental delay incurred at the communication server for jobs originating at host i but transfered for remote service at host j. Relation (2.2) indicates that from host i's point of view the incremental delay incurred for jobs processed at host j, due to the flow from host i to host j, should be equal for all j if there is a positive job flow from host i to host j. On the other hand, if there is no job flow from host i to host j, the incremental delay should be no less than the above value.

Using relation (2.2) Lee and Towsley developed a distributed algorithm for decentralized load balancing. In this algorithm each host compares its own incremental delay to the minimum incremental delay of the other hosts to determine whether to increase or decrease its threshold parameter. The algorithm is iterative in nature and the threshold and transfer probability parameters at each host are updated at each iteration. Initially, each host i sets T_i's and P_{ij}'s to some arbitrary values. At each iteration of the algorithm, host i executes the following.

Algorithm

- Host i computes the incremental delay information $df_i/dx_i(i)$ and $df_i/dx_j(i)$ for $j = 1, 2, ..., N$ and $j \neq i$. It reports the incremental delay information, due to jobs originating at a host $j \neq i$ but transfered for remote service at host i, to every host j, $j = 1, 2, ..., N$.

- Using the incremental delay information

$$df_j/dx_i(j) + dg/dx_i(j) \tag{2.3}$$

  reported by every host $j \neq i$, host i computes the quantity

$$A(i) = \min_{j,j \neq i}\{df_j/dx_i(j) + dg/dx_i(j)\}. \tag{2.4}$$

- Host i compares its own incremental delay to the minimum incremental delay reported by the other hosts:

$$\begin{aligned} &\text{If } df_i/dx_i(i) > A(i) + \theta \quad \Longrightarrow \quad T_i := T_i - 1, \\ &\text{If } df_i/dx_i(i) < A(i) - \theta \quad \Longrightarrow \quad T_i := T_i + 1, \\ &\qquad\qquad \text{else} \quad T_i := T_i, \end{aligned}$$

where θ is a non-negative constant, which must be tuned to prevent threshold change due to a slight imbalance to the incremental delays.

- Update P_{ij}'s using the following formula:

$$P_{ij} = \frac{\mu_j - x_j(j)}{\sum_{k,k\neq i}\{\mu_k - x_k(k)\}} \tag{2.5}$$

where μ_j denotes the maximum processing rate of host j.

In order to implement the above algorithm, the incremental delay information at each host must be computed. We shall assume that the delay incurred due to the transfer of jobs through the communication network is an exponential random variable with parameter μ_c. Therefore, the incremental delay due to the job transfer can be approximated by the mean communication delay μ_c. Recall that each host computer has a communication server that takes care of the job transfers between computers. Consequently, there are two categories of incremental delays; namely, those which are due to local job flow (i.e. $df_i/dx_i(i)$) and those which are due to remote job flow (i.e. $df_j/dx_i(j)$). Incremental delays of the former category are affected by the integer threshold constraints, since local job flow is directly controlled by the threshold parameter. On the other hand, we assume that incremental delays of the latter category are not affected by the integer threshold constraints.

The incremental delay due to local job flow corresponds to the derivative of the mean queue length with respect to the throughput of local jobs (since $x_i(i) = \lambda_i^{(l)}$, Figure 2.2). The following backward difference formula can be used to approximate the incremental delay due to local job flow:

$$\frac{df_i}{dx_i(i)} = \frac{dE[L_i]}{d\lambda_i^{(l)}} \approx \frac{E[L_i]_{T_i} - E[L_i]_{T_i-1}}{[\lambda_i^{(l)}]_{T_i} - [\lambda_i^{(l)}]_{T_i-1}} \tag{2.6}$$

where $E[L_i]_{T_i}$ and $[\lambda_i^{(l)}]_{T_i}$ denote the mean queue length and the local job throughput at host i when the threshold is T_i (Figure 2.2). The purpose of *Estimator-A*, introduced in the next section, is to provide on-line estimates of $E[L_i]_{T_i-1}$ and $[\lambda_i^{(l)}]_{T_i-1}$, given that the system is operating at threshold T_i. The estimates are based on real data gathered from the actual system during an observation interval.

The incremental delay due to remote job flow corresponds to the derivative of the mean queue length at host j with respect to the arrival rate of remote jobs $\lambda_j^{(r)}$ as can be easily seen from relation (2.2) and Figure 2.2. Hence,

$$\frac{df_j}{dx_i(j)} = \frac{dE[L_j]}{d\lambda_j^{(r)}} \qquad (2.7).$$

Estimator-B, described in Section 4, provides on-line estimates of the above gradient information, based on a class of theorems derived from Likelihood Ratios.

3. GRADIENTS WITH RESPECT TO THRESHOLD

3.1. Introduction

In this section we introduce an estimator, whose the purpose is to provide estimates of gradients with respect to the threshold in systems involved in the decentralized threshold scheduling policy described in section 2. The estimator we developed determines the effect of a change in the threshold parameter on the performance metric of interest (i.e. throughput and expected queue length). The assumption is made that either the arrival process or the service time are exponentially distributed. The algorithm uses the *memoryless property* of the exponential distribution in order to efficiently estimate the desired gradients. The major advantage of the proposed estimator is the fact that effectively provides performance sensitivity information while the actual system is running (on-line estimation).

A relatively new approach towards estimating gradients with respect to continuous-valued parameters is referred to as *Perturbation Analysis.* The basis of Perturbation Analysis methodology is extensively described in [9,10]. An extension of this technique to systems where estimation of gradients with respect to integer-valued parameters is desirable, has been attempted in [11], where a Perturbation Analysis algorithm is presented, which provides sensitivity information with respect to a threshold parameter. However, the state memory required by the algorithm in [11] grows to infinity. On the other hand, the estimator introduced in this section requires only four counters in order to provide on-line estimation of gradients with respect to a threshold parameter.

3.2. Gradient Estimation in M/G Systems

We assume that each host computer of the distributed system described in 2.1. can be modeled as a two class $M/G/1/\mathbf{T}$ system, where $\mathbf{T} = (T, \infty)$ since jobs belonging to the second class (remote jobs) are always accepted. Only jobs of the first class (local jobs) are governed by the threshold parameter T. For the sake of simplicity in exposition, we first consider a single class $M/G/1/T$ system with Poisson arrival rate λ and finite queue capacity T, and then we show how the estimator can be very easily extended to a two class system. Let L be the queue length and $TPUT$ the throughput of the system. The physical system will be referred to as the *nominal system*. What we actually need is to estimate the derivative $dE[L]/dTPUT$ while the nominal system is running, using the backward difference formula:

$$\frac{dE[L]}{dTPUT} \approx \frac{E[L]_T - E[L]_{T-1}}{[TPUT]_T - [TPUT]_{T-1}}. \tag{3.1}$$

We shall refer to the system with threshold value $T - 1$ as the *perturbed system*. The nominal system observes itself and after an observation interval estimates the average queue length and the throughput as if it had a threshold value equal to $T - 1$.

Fig. 3.1a illustrates the portion of a nominal sample path for a single class $M/G/1/T$ system, with $T = 3$. Let a_i be the time of the $i-th$ arrival and d_j be the time of the $j-th$ departure. Figure 3.1b represents the corresponding portion of the perturbed system with threshold parameter $T = 2$. We assume that the service time s_k of the $k-th$ customer ($k = 1, 2, ...$) depends only on k. Arrivals a_1 and a_2 are accepted by both the nominal and the perturbed system. However, arrival a_3 is accepted by the nominal system while it is rejected by the perturbed one. In general, every time the nominal system reaches its threshold T the corresponding arrival is shipped out by the perturbed system with threshold value $T - 1$. Obviously, every arrival rejected by the nominal system is also rejected by the perturbed one. As soon as an arrival accepted by the nominal system is rejected by the perturbed one, the latter system has one less customer than the former one. The two systems will continue to differ by one customer until an idle period appears in the nominal path. This is the case with the idle period just before the arrival a_5 in Figures 3.1a and 3.1b.

The first two departures in both the nominal and the perturbed paths of Figure 3.1a and 3.1b occur at times d_1 and d_2 ($d_1 = d_1^p$, $d_2 = d_2^p$). Departure d_2 leaves the nominal system with just one customer. Since the perturbed system has one less customer than the nominal one (due to the rejection of arrival a_3), a new *idle period* will appear in the perturbed path. This idle period is terminated by arrival a_4.

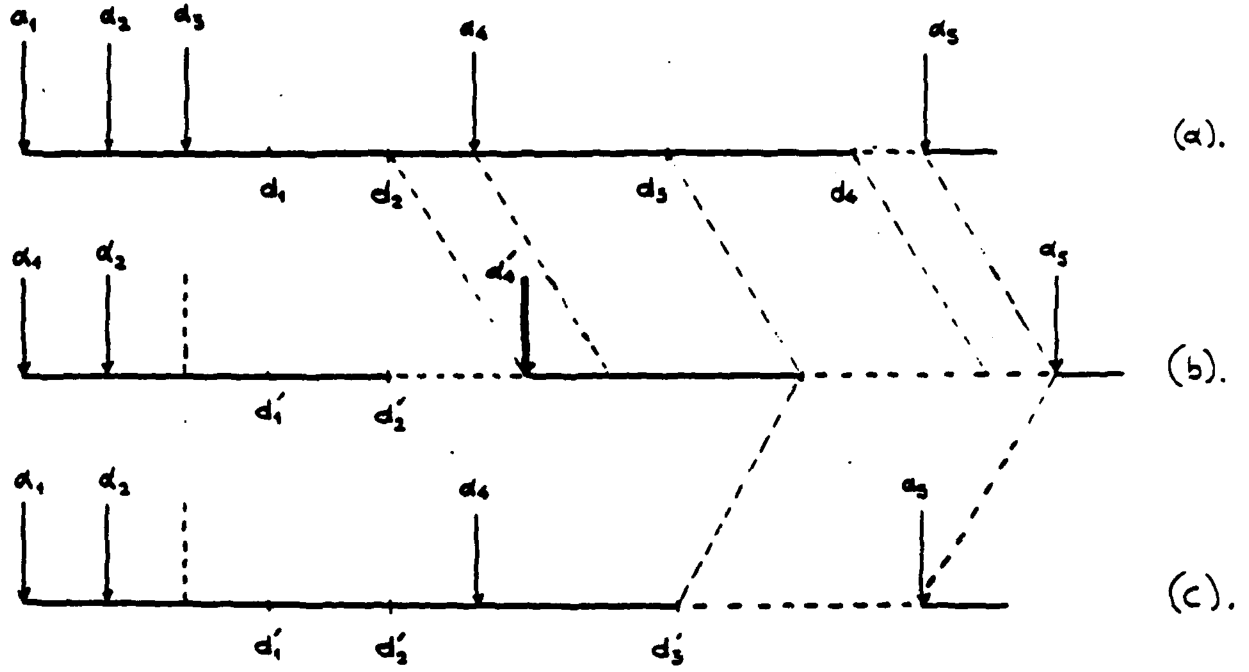


Fig. 3.1 - Estimation of gradients with respect to the threshold.

Therefore, the major effect of the perturbation $\delta T = -1$ on the perturbed path is the generation of new idle periods, due to the rejection of arrivals which are accepted by the nominal system. Since all the jobs rejected by the nominal system are also rejected by the perturbed one, it is not possible that an idle period that appeared in the nominal path be eliminated in the perturbed path.

The nominal system observes itself and can trace the case where the nominal and perturbed paths are out of phase (i.e. their queue lengths differ by one). This happens every time an arrival accepted by the nominal system is rejected by the perturbed one. When a departure leaves the nominal system with only one job and the two systems are out of phase, then the nominal system knows that the perturbed one starts an idle period waiting for the next customer. Instead of waiting for that next arrival, we generate an *idle period* terminated by a *ficticious customer* and we assign to that customer the currently available service time. This new idle period is derived from an exponential distribution with parameter λ; this is permissible because of the *memoryless property* of the Poisson arrival process. All the subsequent events in the perturbed path are shifted in time by the introduced idle period. The above situation is illustrated in Figure 3.1c, where a_4' is the ficticious arrival terminating the introduced idle period.

Actually, although the sample path shows a specific idle period length, it is not required by the algorithm performing the estimation. We only need to keep a counter of how many idle periods have been introduced in the perturbed system during the observation interval. During that interval the nominal system can estimate the rate λ_{est} of its arrival process. At the end of the observation interval, it can use this value to estimate the *total idle time* introduced in the perturbed system, using the formula:

$$idle\ time = \frac{(\#\ idle\ periods)}{\lambda_{est}}. \tag{3.2}$$

In the following we describe the proposed *Estimator-A* more formally. We make the assumptions that the arrival process is a Poisson process and that the nominal system monitors its average queue length $E[L]$ and its throughput $TPUT$. The following four counters are required by the algorithm:

- ***PHASE***: It indicates whether the nominal and the perturbed system are out of phase. It is 0 when both systems have the same number of customers; otherwise it is 1.
- ***PLAST***: It indicates the last time the $PHASE$ changed from 0 to 1.
- ***TLESS***: Total time during the observation interval that the perturbed system has one less customer than the nominal one.
- ***IDLE***: Total number of idle periods introduced in the perturbed path during the observation interval.

Estimator-A

- Initially the counters $PHASE, PLAST, TLESS$ and $IDLE$ are set to 0.
- At the $i-th$ arrival the following instructions are executed:

```
If ( $L(a_i) = T$ ) and ( $PHASE = 0$ ) then
   begin
      $PHASE := 1;$
      $PLAST := a_i;$
   end
```

  where a_i denotes the time of the $i-th$ arrival and $L(a_i)$ the number of customers right after the $i-th$ arrival.

- At the $j-th$ departure the following instructions are executed:

```
If ( $L(d_j) = 1$ ) and ( $PHASE = 1$ ) then
   begin
      $IDLE := IDLE + 1;$
      $PHASE := 0;$
      $TLESS := TLESS + d_j - PLAST;$
   end
```

where d_j denotes the time of the $j-th$ departure and $L(d_j)$ the number of customers right after the $j-th$ departure.

The nominal system observes itself for an observation interval τ. At the end of that interval knowing the statistics $E[L]_T$ and $[TPUT]_T$, it is able to estimate the corresponding statistics for the perturbed system using the following formulas:

$$[TPUT]_{T-1} = \frac{\tau[TPUT]_T}{\tau + (IDLE/\lambda_{est})} \tag{3.3}$$

$$E[L]_{T-1} = E[L]_T - \frac{TLESS + (IDLE/\lambda_{est})}{\tau + (IDLE/\lambda_{est})} \tag{3.4}$$

where $IDLE/\lambda_{est}$ denotes the total idle time introduced in the perturbed path. The minimum length of the observation interval τ necessary to obtain a good estimate for $E[L]_{T-1}$ and $[TPUT]_{T-1}$ depends on the specific application of the system and can be determined empirically. As it turns out from the simulation results it depends on the utilization of the system. Section 3.3. contains simulation results to demonstrate the performance of the proposed estimator and discusses the convergence properties of the algorithm.

Estimator-A can be very easily used in the case of an $M/G/1/\mathbf{T}$ system with two classes of jobs (local and remote), where $\mathbf{T} = (T, \infty)$. Note that this is the model of the host computers, in the decentralized threshold scheduling policy described in Section 2. Remote jobs are always accepted by both the nominal and the perturbed system. Therefore, only local jobs can be accepted by the nominal system and rejected by the perturbed one, as a result of the change in the threshold value. Let λ_{est} be the arrival rate of jobs coming in from the external world and $\lambda_{est}^{(r)}$ be the arrival rate of remote jobs, estimated during the observation interval. Then the *idle time* introduced in the perturbed system is given by the formula:

$$idle\ time = \frac{IDLE}{\lambda_{est} + \lambda_{est}^{(r)}}. \tag{3.5}$$

The rest of the algorithm works exactly in the same way as in the single class $M/G/1/T$ system.

3.3. Simulation Results

In this section, we present simulation results to demonstrate the performance of *Estimator-A*. As an example, Tables 3.1 through 3.6 contain simulation results for a single class $M/M/1/3$ system with service rate $\mu = 1.0$ *jobs/sec* and different values of arrival rate: $\lambda = 0.2, 0.5$ and 0.8 *jobs/sec*. *Estimator-A* is used to provide on-line estimates of the average queue length $E[L]$ and throughput $TPUT$ of the perturbed system with threshold $T = 2$. The estimated values are compared to the actual ones which are computed analytically. In order to point out the convergence properties of the estimation technique, the simulation results are presented for different observation intervals.

Tablés 3.7 and 3.8 contain simulation results for an $M/H_2/1/3$ system with one class of jobs, hyperexponential service time with parameters $\mu = 1.0$ and $C_v = 2$ and Poisson arrivals with parameter $\lambda = 0.5$ *jobs/sec*. The "actual" values of $E[L]_{T=2}$ and $[TPUT]_{T=2}$ are obtained by simulating the corresponding system with threshold $T = 2$.

#completions	$[TPUT]^{act}_{T=2}$	$[TPUT]^{est}_{T=2}$	*%error*	*#runs*	σ_e
50	0.426	0.450	5.33	10	0.064
100	0.426	0.440	3.28	10	0.042
200	0.426	0.438	2.81	10	0.035
500	0.426	0.430	0.94	10	0.015
1000	0.426	0.426	0.00	10	0.007
2000	0.426	0.426	0.00	10	0.007
5000	0.426	0.426	0.00	10	0.003
10000	0.426	0.426	0.00	10	0.003

Table 3.1 - Estimation of $[TPUT]_{T=2}$ for an $M/M/1/3$ system with $\lambda/\mu = 0.2$.

#completions	$E[L]^{act}_{T=2}$	$E[L]^{est}_{T=2}$	%error	#runs	σ_x
50	0.571	0.599	4.90	10	0.114
100	0.571	0.590	3.33	10	0.093
200	0.571	0.553	3.15	10	0.069
500	0.571	0.554	2.97	10	0.039
1000	0.571	0.560	1.92	10	0.021
2000	0.571	0.564	1.22	10	0.015
5000	0.571	0.565	1.05	10	0.006
10000	0.571	0.567	0.70	10	0.005

Table 3.2 - Estimation of $E[L]_{T=2}$ for an $M/M/1/3$ system with $\lambda/\mu = 0.2$.

#completions	$[TPUT]^{act}_{T=2}$	$[TPUT]^{est}_{T=2}$	%error	#runs	σ_x
50	0.194	0.212	9.27	10	0.042
100	0.194	0.201	3.61	10	0.027
200	0.194	0.199	2.57	10	0.019
500	0.194	0.195	0.52	10	0.006
1000	0.194	0.194	0.00	10	0.004
2000	0.194	0.194	0.00	10	0.003
5000	0.194	0.194	0.00	10	0.002
10000	0.194	0.194	0.00	10	0.002

Table 3.3 - Estimation of $[TPUT]_{T=2}$ for an $M/M/1/3$ system with $\lambda/\mu = 0.5$.

#completions	$E[L]^{act}_{T=2}$	$E[L]^{est}_{T=2}$	*%error*	*#runs*	σ_x
50	0.225	0.239	6.22	10	0.065
100	0.225	0.220	2.22	10	0.044
200	0.225	0.217	3.55	10	0.029
500	0.225	0.219	2.66	10	0.014
1000	0.225	0.220	2.22	10	0.010
2000	0.225	0.221	1.77	10	0.005
5000	0.225	0.222	1.33	10	0.003
10000	0.225	0.222	1.33	10	0.003

Table 3.4 - Estimation of $E[L]_{T=2}$ for an $M/M/1/3$ system with $\lambda/\mu = 0.5$.

#completions	$[TPUT]^{act}_{T=2}$	$[TPUT]^{est}_{T=2}$	*%error*	*#runs*	σ_x
50	0.590	0.644	9.28	10	0.091
100	0.590	0.602	2.12	10	0.055
200	0.590	0.600	1.70	10	0.046
500	0.590	0.595	0.84	10	0.021
1000	0.590	0.591	0.17	10	0.009
2000	0.590	0.590	0.00	10	0.006
5000	0.590	0.590	0.00	10	0.006
10000	0.590	0.590	0.00	10	0.003

Table 3.5 - Estimation of $[TPUT]_{T=2}$ for an $M/M/1/3$ system with $\lambda/\mu = 0.8$.

#completions	$E[L]^{act}_{T=2}$	$E[L]^{est}_{T=2}$	*%error*	*#runs*	σ_s
50	0.855	0.943	10.29	10	0.147
100	0.855	0.901	5.38	10	0.069
200	0.855	0.880	2.92	10	0.055
500	0.855	0.874	2.22	10	0.035
1000	0.855	0.868	1.52	10	0.020
2000	0.855	0.865	1.17	10	0.016
5000	0.855	0.862	0.82	10	0.011
10000	0.855	0.860	0.58	10	0.008

Table 3.6 - Estimation of $E[L]_{T=2}$ for an $M/M/1/3$ system with $\lambda/\mu = 0.8$.

#completions	$[TPUT]^{act}_{T=2}$	$[TPUT]^{est}_{T=2}$	*%error*	*#runs*	σ_s
50	0.406	0.460	13.30	10	0.114
100	0.406	0.413	1.72	10	0.058
200	0.406	0.417	2.70	10	0.032
500	0.406	0.412	1.47	10	0.020
1000	0.406	0.404	0.49	10	0.012
2000	0.406	0.406	0.00	10	0.007
5000	0.406	0.407	0.24	10	0.007
10000	0.406	0.406	0.00	10	0.004

Table 3.7 - Estimation of $[TPUT]_{T=2}$ for an $M/H_2/1/3$ system with $\lambda/\mu = 0.5$, $C_v = 2$.

#completions	$E[L]^{act}_{T=2}$	$E[L]^{est}_{T=2}$	%error	#runs	σ_e
50	0.593	0.605	2.02	10	0.155
100	0.593	0.592	0.17	10	0.067
200	0.593	0.602	1.52	10	0.056
500	0.593	0.584	1.54	10	0.054
1000	0.593	0.580	2.19	10	0.044
2000	0.593	0.584	1.54	10	0.025
5000	0.593	0.584	1.54	10	0.023
10000	0.593	0.588	0.84	10	0.015

Table 3.8 - Estimation of $E[L]_{T=2}$ for an $M/H_2/1/3$ system with $\lambda/\mu = 0.5$, $C_v = 2$.

Tables 3.9 and 3.10 contain simulation results for an $M/M/1/(3,\infty)$ system with two classes of jobs, where only jobs belonging to the first class are governed by the threshold parameter. Note that this is the model of each of the host computers involved in the decentralized scheduling policy described in Section 2. In our experiment the arrival rate of the first class (jobs coming in from the external world) is $\lambda = 0.5$ *jobs/sec*, the arrival rate of the second class (remote jobs) is $\lambda^{(r)} = 0.25$ *jobs/sec* and the service rate is $\mu = 1.0$ *jobs/sec*. The estimated parameters are the local throughput $[TPUT^{(l)}]_{T=2}$ and the average queue length $E[L]_{T=2}$ of the perturbed system.

#completions	$[TPUT^{(l)}]^{act}_{T=2}$	$[TPUT^{(l)}]^{est}_{T=2}$	%error	#runs	σ_e
100	0.350	0.365	4.28	10	0.167
500	0.350	0.365	4.28	10	0.017
1000	0.350	0.365	4.28	10	0.016
5000	0.350	0.360	2.86	10	0.004
10000	0.350	0.356	1.71	10	0.004

Table 3.9 - Estimation of $[TPUT^{(l)})]_{T=2}$ for a 2-class $M/M/1/(3,\infty)$ system.

#completions	$E[L]^{act}_{T=2}$	$E[L]^{est}_{T=2}$	*%error*	*#runs*	σ_x
100	1.000	0.670	33.00	10	0.121
500	1.000	1.055	5.50	10	0.110
1000	1.000	1.041	4.10	10	0.041
5000	1.000	1.016	1.60	10	0.023
10000	1.000	1.015	1.50	10	0.019

Table 3.10 - Estimation of $E[L]_{T=2}$ for a 2-class $M/M/1/(3,\infty)$ system.

3.4. Gradient Estimation in G/M Systems

Estimator-A can be used to provide gradient estimates for G/M systems. In this case we take advantage of the *memoryless property* of the service time distribution. The nominal system observes itself for an observation interval τ and during that interval it counts the number of jobs rejected by the perturbed system, as a result of the perturbation $\delta T = -1$ in the threshold parameter. We assume also that the nominal system can monitor its throughput and average queue length. Four counters are required for the implementation of the algorithm; namely $PHASE$, $TLESS$, $PLAST$ and $NREJ$. The first three counters have exactly the same meaning as in the M/G case, while the fourth denotes the number of jobs rejected by the perturbed system but accepted by the nominal one. The algorithm can be described as follows:

- Initially the counters $PHASE$, $PLAST$, $TLESS$ and $NREJ$ are set to 0.
- At the $i-th$ arrival the following instructions are executed:

```
If ( $L(a_i) = T$ ) and ( $PHASE = 0$ ) then do
    begin
        $PHASE := 1$;
        $PLAST := a_i$;
        $NREJ := NREJ + 1;$
    end
```

where a_i denotes the time of the $i-th$ arrival and $L(a_i)$ the number of customers right after the $i-th$ arrival.

- At the $j-th$ departure the following instructions are executed:

```
If ( $L(d_j) = 1$ ) and ( $PHASE = 1$ ) then do
    begin
        $TLESS := TLESS + d_j - PLAST;$
        $PHASE := 0;$
    end
```

where d_j denotes the time of the $j-th$ departure and $L(d_j)$ the number of customers right after the $j-th$ departure.

We can compute the estimated quantities by using formulas analogous to those of the M/G case. For example the $[TPUT]_{T-1}$ of the perturbed system can be calculated as follows:

$$[TPUT]_{T-1} = [TPUT]_T - \frac{NREJ}{\tau} \qquad (3.6)$$

where τ is the observation interval.

4. GRADIENTS WITH RESPECT TO THE ARRIVAL RATE

4.1. Introduction

In this section we introduce an estimator, whose the purpose is to estimate performance sensitivities with respect to the arrival rate in systems involved in the decentralized threshold scheduling policy described in section 2. The estimator we have developed determines the derivative of mean values (e.g. average queue length) with respect to an arrival rate. The method is based on the work done by Reiman and Weiss [12], who used *likelihood ratio* techniques to prove their theorems. This is a typical result: if $E_\lambda(\psi)$ is the mean value of quantity ψ as a function of a Poisson rate λ then

$$\frac{d}{d\lambda}E_\lambda(\psi) = E_\lambda[(\frac{N}{\lambda} - T)\psi] \tag{4.1}$$

where T is the duration of the observation interval and N is the number of Poisson events in time T [11]. Therefore, using the idea that derivatives of expectations are themselves expectations, we can have a consistent estimate of $dE_\lambda(\psi)/d\lambda$ by simply estimating $E[(N/\lambda - T)\psi]$ during the observation period T. Reiman and Weiss assume in their work that the Poisson rate λ is known; we have slightly modify their method so that the Poisson rate λ can be estimated during the observation interval T. The method is extremely well suited to regenerative systems and can be implemented with very little increase in running time and memory requirements. Note also that the method is suitable for any parameter (not only Poisson rates) which does not change the possible sample paths, but merely changes their probability [12]. We will use this method to estimate the incremental delay due to remote job flow at host i of the distributed system considered throughout this work.

4.2. Gradient Estimation in Regenerative Systems

In this section we will show how the theorems derived in [11] can be used in *regenerative systems* for automatically estimating derivatives. Our discussion is based on regenerative process theory which shows that steady state expected values can be obtained as ratios of expected values over a regenerative cycle. We assume that our queueing system can be modeled as an $M/G/1$ system with Poisson arrival rate λ.

We consider the initiation of a new busy cycle as the regeneration point. Let 0 be the number of customer initiating the $i-th$ busy cycle, N_i be the number of the first customer to encounter an empty system (indicating the initiation of the $(i+1)-th$ busy cycle) and T_i be the duration of the $i-th$ busy cycle. Assume that we observe the system for a number of busy cycles and W_i is the total waiting time in the $i-th$ busy period. We then have:

$$E[L] = \frac{E[W]}{E[T]} \tag{4.2}$$

where E[L] is the average queue length over the total observation period. What we really want to compute, is the quantity $dE[L]/d\lambda$. The quotient rule combined with (4.2) yields

$$\frac{dE[L]}{d\lambda} = \frac{\frac{dE[W]}{d\lambda}E[T] - \frac{dE[T]}{d\lambda}E[W]}{E[T]^2}. \tag{4.3}$$

Now we can use the results derived by Reiman and Weiss [11] to represent the derivatives appearing on the right hand side of (4.3) as

$$\frac{d}{d\lambda}E[W] = E[(\frac{N}{\lambda} - T)W] \tag{4.4}$$

and

$$\frac{d}{d\lambda}E[T] = E[(\frac{N}{\lambda} - T)T] \tag{4.5}$$

The expectations at the right hand side of equations (4.4) and (4.5) can be very easily computed as averages over n busy cycles. Thus,

$$E[(\frac{N}{\lambda} - T)W] = \frac{1}{n}\sum_{i=1}^{n}(\frac{N_i}{\lambda} - T_i)W_i \tag{4.6}$$

$$E[(\frac{N}{\lambda} - T)T] = \frac{1}{n}\sum_{i=1}^{n}(\frac{N_i}{\lambda} - T_i)T_i \qquad (4.7).$$

Therefore, by substituting (4.6) and (4.7) in (4.3) we have

$$\frac{d}{d\lambda}E[L] = \frac{[\sum_{i=1}^{n} T_i][\sum_{i=1}^{n}(\frac{N_i}{\lambda} - T_i)W_i] - [\sum_{i=1}^{n} W_i][\sum_{i=1}^{n}(\frac{N_i}{\lambda} - T_i)T_i]}{[\sum_{i=1}^{n} T_i]^2} \qquad (4.8).$$

As we can easily see from equation (4.8) we only need to keep track of N_i, T_i and W_i and update the corresponding counter at the end of each busy cycle.

Although the above analysis implies that the Poisson arrival rate is known, we can slightly modify it so that it can be applied to a system which estimates its arrival rate simultaneously with its evolution. Actually, we only need to update the following counters at the end of each busy cycle:

$$C_1 = \sum_{i=1}^{n} T_i, \qquad C_2 = \sum_{i=1}^{n} W_i, \qquad C_3 = \sum_{i=1}^{n} N_i W_i,$$

$$C_4 = \sum_{i=1}^{n} T_i W_i, \qquad C_5 = \sum_{i=1}^{n} (T_i)^2, \qquad C_6 = \sum_{i=1}^{n} N_i T_i.$$

At the end of the observation period (i.e. after n busy cycles) we can use the above counters and the estimated arrival rate λ_{est} to compute the derivative of the average queue length with respect to the arrival rate as follows:

$$\frac{d}{d\lambda}E[L] = \frac{C_1(\frac{C_3}{\lambda_{est}} - C_4) - C_2(\frac{C_6}{\lambda_{est}} - C_5)}{(C_1)^2}. \qquad (4.9)$$

A similar analysis can be applied for estimating derivatives of other mean values with respect to a Poisson rate. As an example, we refer to the estimation of the derivative of the average response time $E[R]$ of a job with respect to the arrival rate λ using the formula

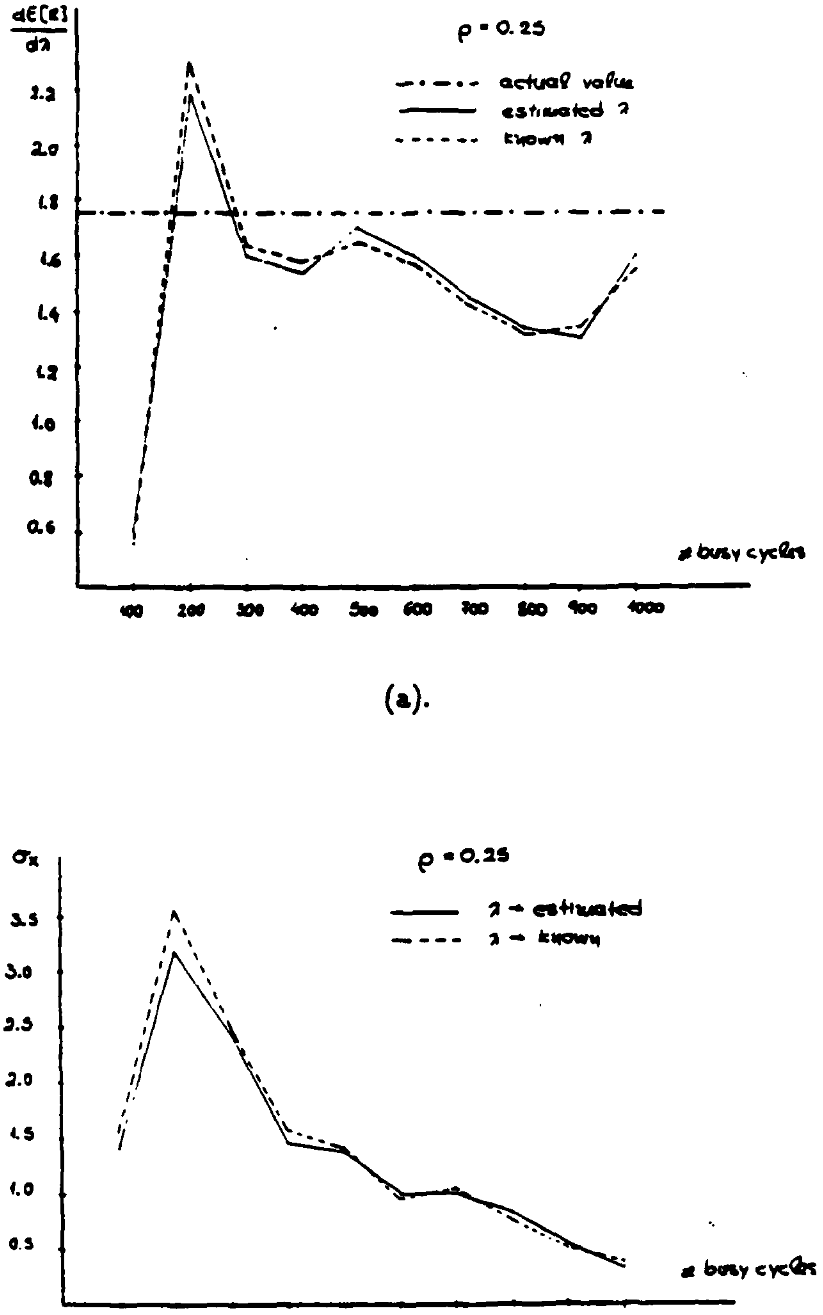
$$E[R] = \frac{E[W]}{E[N]} \qquad (4.10)$$

where $E[W]$ is the average total waiting time and $E[N]$ is the average number of arrivals over n busy cycles.

4.3. Simulation Results

To investigate the performance and convergence of the sensitivity analysis method described in the previous sections, we applied it to a regenerative simulation of an $M/M/1$ system with a single class of jobs. The choice of an $M/M/1$ queue was made because the simulation itself was very easy to write, and theoretical results are available, making it possible to check the numerical results. We investigate the performance of the estimation algorithm for three different values of the utilization, namely $\rho = 0.25$, 0.5 and 0.9, and for different lengths of the observation period. We also compare the convergence of the original algorithm which considers a known arrival rate to the convergence of the modified algorithm which uses the estimated arrival rate over the observation period. As it turns out from the simulation results the modified algorithm behaves as good as the original one.

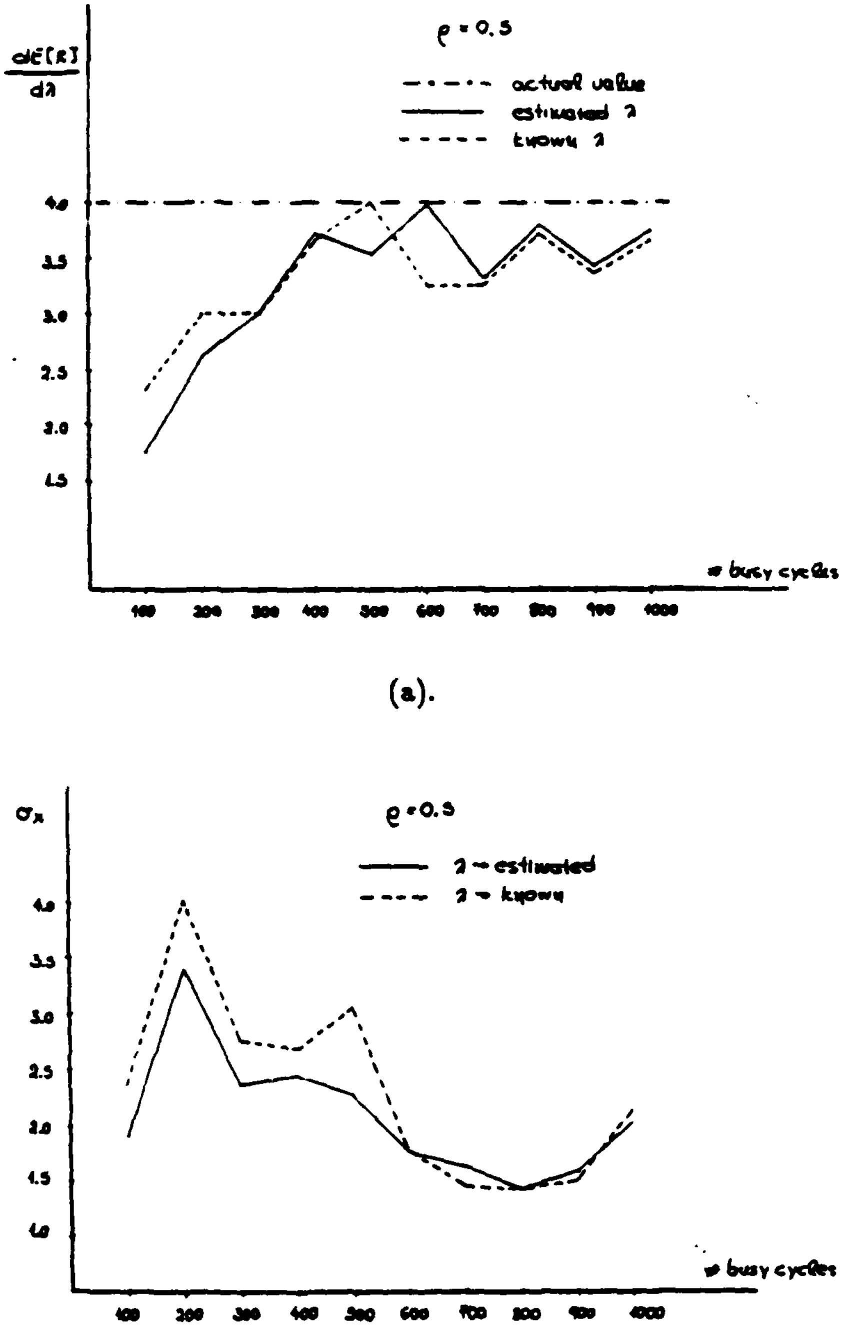
Figures 4.1 to 4.6 show the behavior of the algorithm for the three different values of utilization mentioned above with respect to the length of the observation interval (in number of busy cycles). The first three figures (4.1 to 4.3) present simulation results for observation intervals in the range 100 to 1000 busy cycles. Figures 4.4 to 4.6 contain the corresponding results for observation periods in the range 1000 to 50000 busy cycles. Each figure consists of two parts. Part-a shows the estimated derivative of the average response time of a job with respect to the arrival rate λ for both the algorithms with known and estimated arrival rate. The resulting data points are the average over 20 runs. Part-b shows the standard deviation over the 20 runs.



(a).

(b).

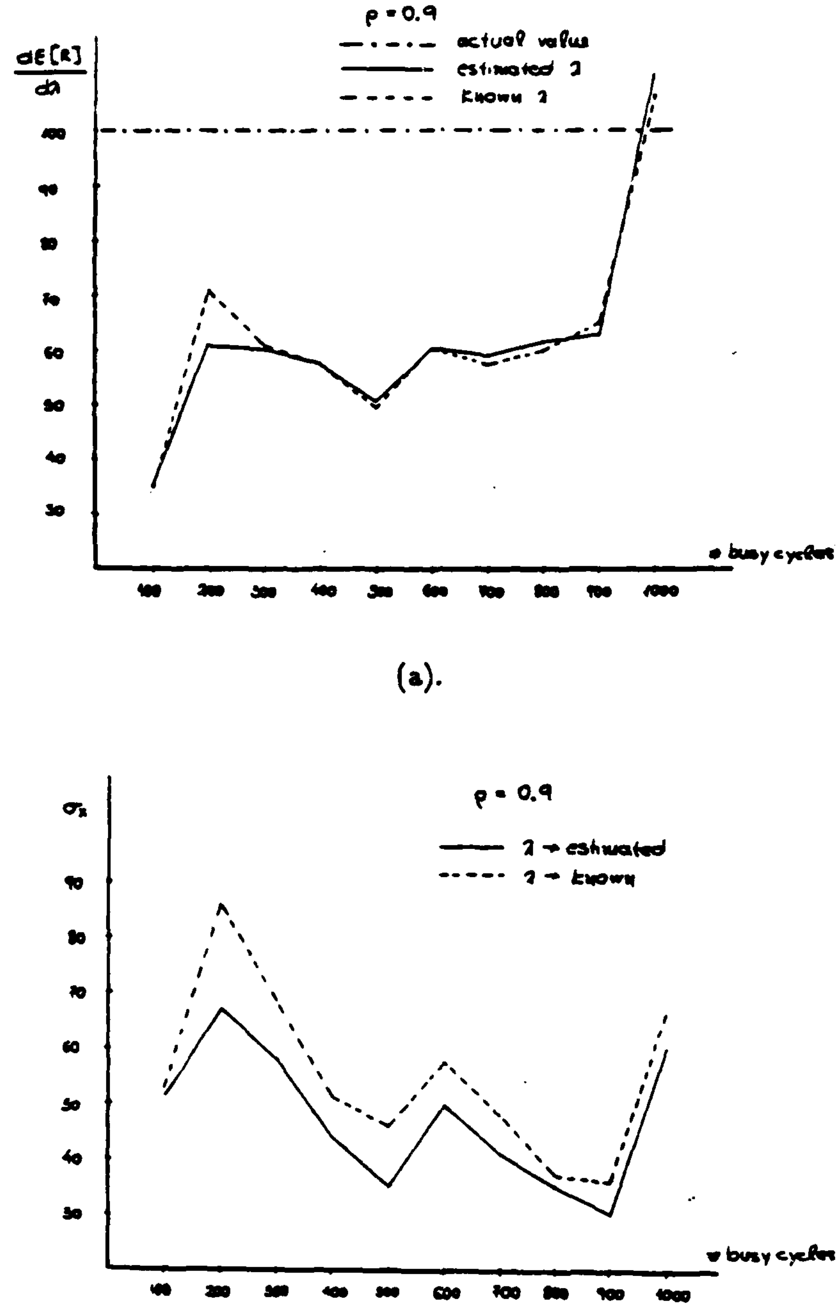
Fig. 4.1 - Estimation of $dE[R]/d\lambda$ for an $M/M/1$ system with $\rho = 0.25$ (observation period: 100 to 1000 busy cycles).



(a).

(b).

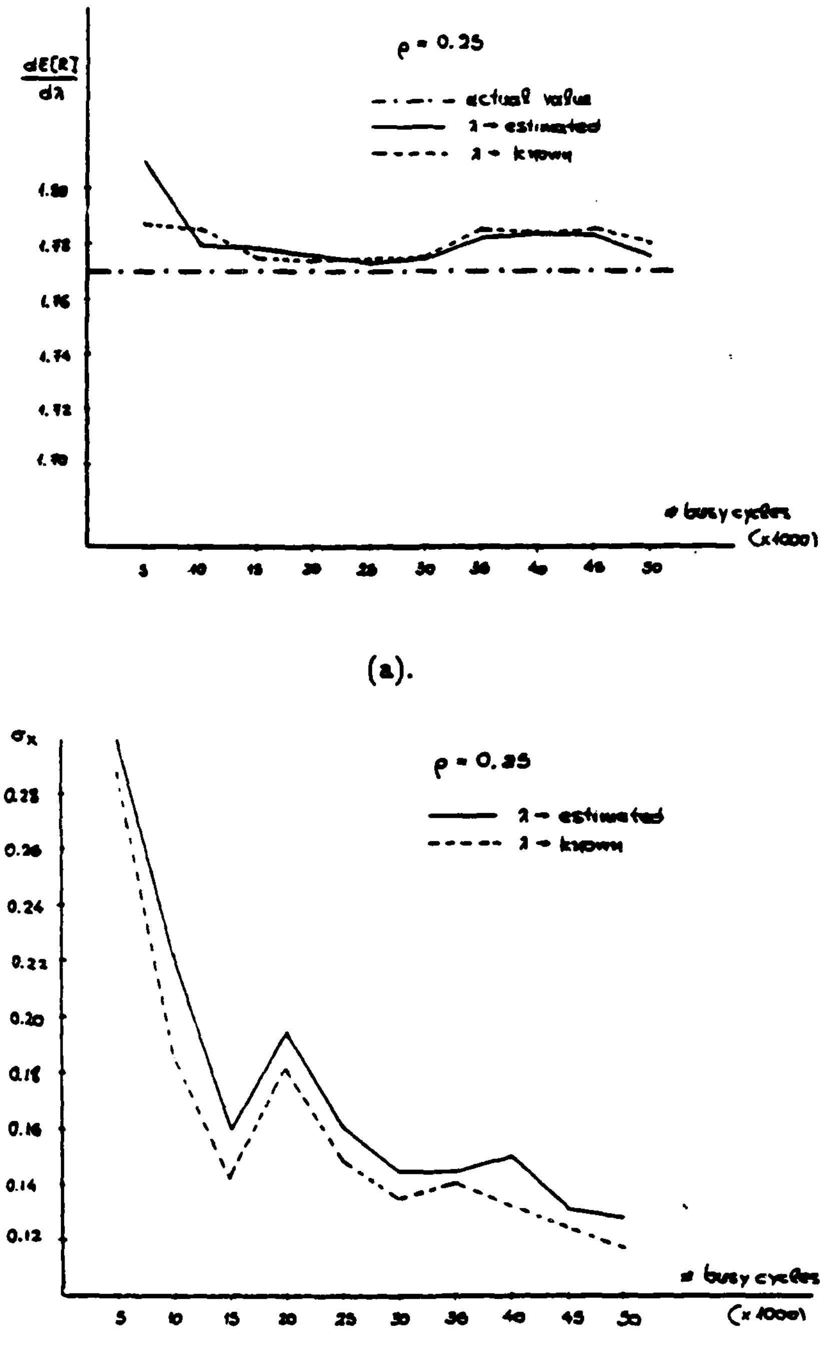
Fig. 4.2 - Estimation of $dE[R]/d\lambda$ for an $M/M/1$ system with $\rho = 0.5$ (observation period: 100 to 1000 busy cycles).



(a).

(b).

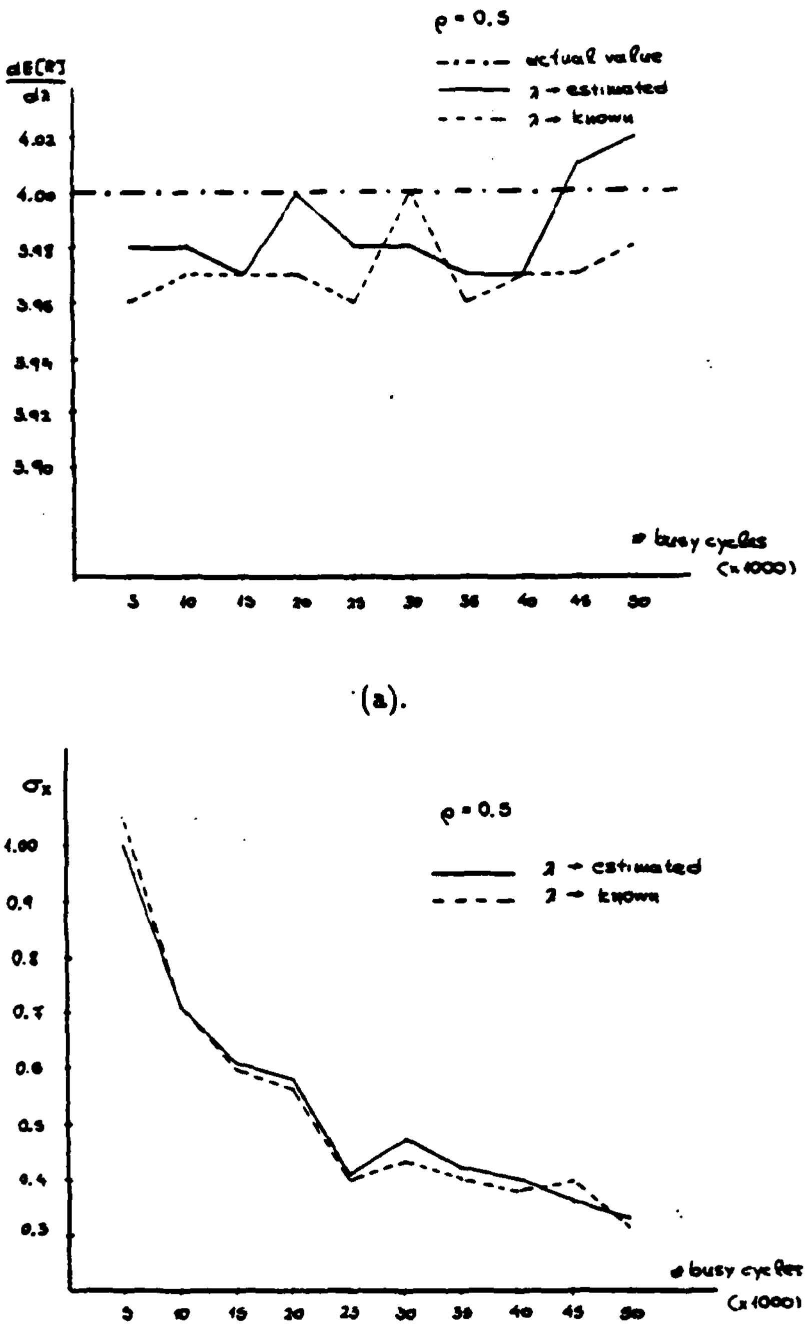
Fig. 4.3 - Estimation of $dE[R]/d\lambda$ for an $M/M/1$ system with $\rho = 0.9$ (observation period: 100 to 1000 busy cycles).



(a).

(b).

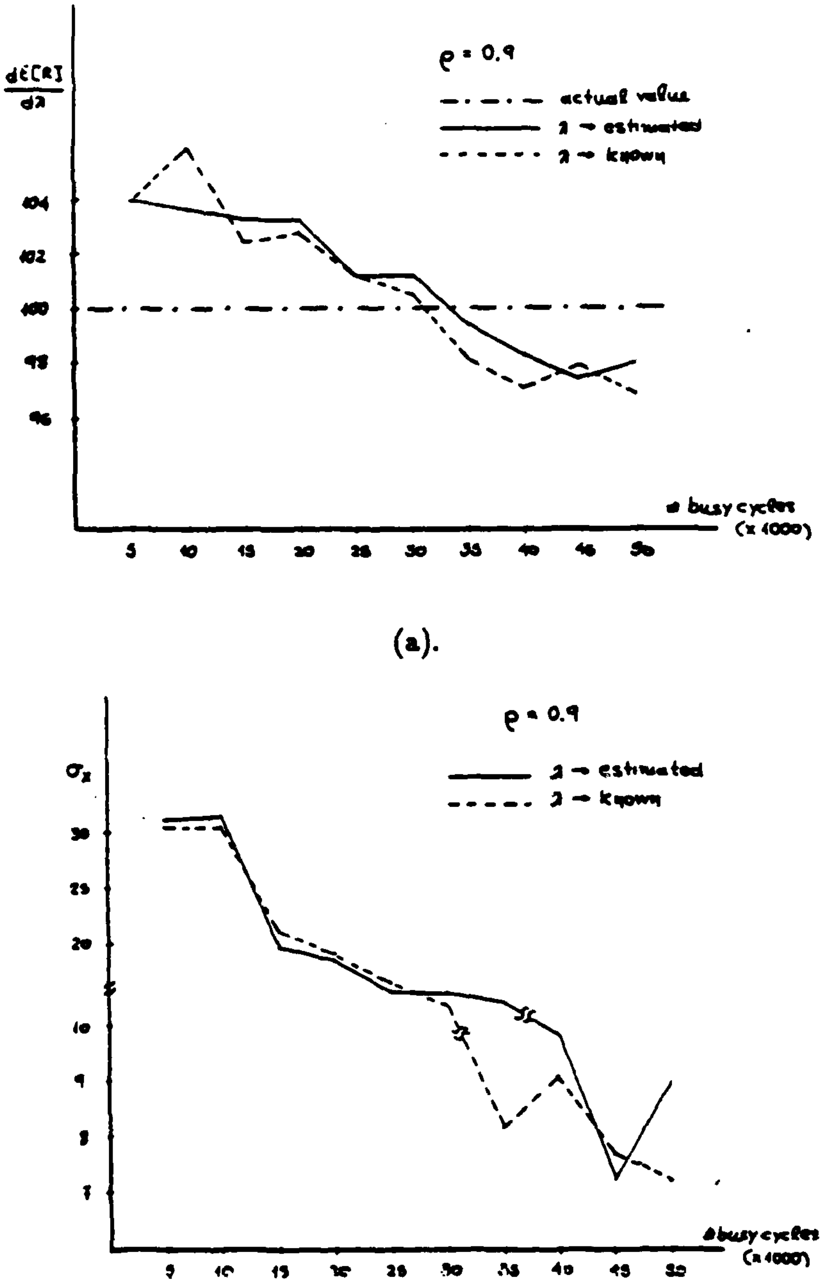
Fig. 4.4 - Estimation of $dE[R]/d\lambda$ for an $M/M/1$ system with $\rho = 0.25$ (observation period: 1000 to 50000 busy cycles).



(a).

(b).

Fig. 4.5. - Estimation of $dE[R]/d\lambda$ for an $M/M/1$ system with $\rho = 0.5$ (observation period: 1000 to 50000 busy cycles).



(a).

(b).

Fig. 4.6 - Estimation of $dE[R]/d\lambda$ for an $M/M/1$ system with $\rho = 0.9$ (observation period: 1000 to 50000 busy cycles).

The actual values of $dE[R]/d\lambda$ for the different utilizations, indicated in all the Figures, can be computed very easily. We can notice that the observation period must be long enough in order for the algorithm to give consistent estimates of the desired gradient. If the observations take place over a short period of time, the system does not have the time to respond to the arrivals and furthermore, the variability of arrivals is relatively high over short periods of time. However, we believe that the method can be effectively applied in the distributed threshold scheduling policy described in Section 2. The reason is that we only need to compare derivatives without worrying about their absolute value. Therefore, the estimate derived even over a small number of busy cycles (e.g. 200 cycles or less) is probably good enough for that purpose.

4.4. Systems with two classes of jobs

So far, we have discussed a method (Estimator-B) for estimating derivatives of mean values with respect to a Poisson arrival rate in systems with one class of jobs. However, the systems involved in the decentralized threshold scheduling policy adopted in this work have two different classes of arrivals; namely local and remote jobs. In this section we show that we can very easily extend the method described in section 4.2 so that we are able to estimate the gradient of the average queue length with respect to the arrival rate of remote jobs. We recall that this gradient represents the incremental delay due to remote job flow and is required by the threshold updating algorithm described in section 2.3.

We assume that each host computer of the distributed system described in 2.1. can be modeled as a two class $M/G/1/\mathbf{T}$ system, where $\mathbf{T} = (T, \infty)$ since jobs belonging to the second class (remote jobs) are always accepted. Only local jobs are governed by the threshold parameter T. The purpose of *Estimator-B* is to estimate the gradient $dE[L]/d\lambda^{(r)}$, where $\lambda^{(r)}$ is the arrival rate of remote jobs. Using equation (4.2) and the quotient rule we have

$$\frac{dE[L]}{d\lambda^{(r)}} = \frac{\frac{dE[W]}{d\lambda^{(r)}}E[T] - \frac{dE[T]}{d\lambda^{(r)}}E[W]}{E[T]^2}. \qquad (4.11)$$

Using equation (4.1) we can represent the derivatives in the right hand side of (4.11) as

$$\frac{d}{d\lambda^{(r)}}E[W] = E[(\frac{N^{(r)}}{\lambda^{(r)}} - T)W] \tag{4.12}$$

and

$$\frac{d}{d\lambda^{(r)}}E[T] = E[(\frac{N^{(r)}}{\lambda^{(r)}} - T)T] \tag{4.13}$$

where $N^{(r)}$ corresponds to the number of remote arrivals and $\lambda^{(r)}$ to the arrival rate of remote jobs. All the other variables have the same meaning as in equations (4.4) and (4.5). The same method as in section 4.2 can be used to compute the expectations in the right hand side of equations (4.12) and (4.13). We can also use the same modification as in section 4.2 to apply the method to a system which estimates the arrival rate of remote jobs while it runs.

Table 4.1 contains simulation results for an $M/M/1/(3,\infty)$ system with two classes of jobs, where only jobs belonging to the first class are governed by the threshold parameter. In our experiment the arrival rate of the first class (jobs coming in from the external world) is $\lambda = 0.5$ *jobs/sec*, the arrival rate of the second class (remote jobs) is $\lambda^{(r)} = 0.25$ *jobs/sec* and the service rate is $\mu = 1.0$ *jobs/sec*. Estimator-B is used to estimate the derivative of the average queue length with respect to the arrival rate of remote jobs. It can be derived analytically that the actual value of the above derivative for the particular system is $dE[L]/d\lambda^{(r)} = 2.70$.

#busy cycles	$[dE[L]/d\lambda^{(r)}]^{act}$	$[dE[L]/d\lambda^{(r)}]^{est}$	*%error*	*#runs*	σ_e
1000	2.700	2.446	9.40	5	0.277
5000	2.700	2.748	1.77	5	0.200
10000	2.700	2.690	0.37	5	0.157
20000	2.700	2.693	0.26	5	0.134

Table 4.1 - Estimation of $dE[L]/d\lambda^{(r)}$ for a 2-class $M/M/1/(3,\infty)$ system.

5. AN EXAMPLE

In this section we consider a simple example where host computers are interconnected through a communication network and execute the *distributed load balancing* algorithm described in section 2. The estimation techniques described in sections 3 and 4 are imbedded in the decentralized threshold scheduling policy to provide the necessary gradient information. Each host computer is modeled as a single-server queueing system with two classes of arrivals; namely local and remote jobs. In such a system we study the behavior of the algorithm in a static as well as in a changing environment.

The service time at host i is exponentially distributed with mean $1/\mu_i$ for $i = 1, 2, ..., N$. The interarrival times of jobs coming in a host from the external world are exponentially distributed with mean $1/\lambda_i$ for $i = 1, 2, ..., N$. The communication delay of a job due to its transfer for remote service is assumed to be an exponentially distributed random variable with mean $1/\mu_c$. Recall that jobs can move only once through the communication network, since remote jobs are always accepted. Whenever a host rejects a job because it reaches its threshold, it sends this job randomly to any one of the other hosts for remote service. This last assumption is reasonable in the case where all the hosts are homogeneous with the same utilization.

We have simulated a distributed system with five host computers which can be easily expanded to a system with N hosts. Each host executes the threshold updating algorithm independently to each other. All of them start the execution of the algorithm at time 0. Initial thresholds values for each host are drawn randomly from a uniform distribution. Each host observes itself for a number of busy cycles and at the end of the observation period computes the derivatives of its average queue length with respect to the local and remote job flow. Then, it sends the necessary gradient information to the other nodes and uses the currently available gradients reported by the other nodes, as well as its own gradient information to update its local threshold. Threshold updating completes an iteration of the algorithm at that node. Another observation period can start immediately and the host computer will execute the balancing policy with the new threshold value until the next iteration. Each host computer has a communication server that takes care of the job transfers between computers. The incremental delay due to job transfers, required by the threshold updating algorithm, is approximated by the average communication delay μ_c.

We can make the following interesting observation regarding the application of the estimation techniques discussed in sections 3 and 4 in the decentralized threshold scheduling policy. If the initial threshold value at a particular node is large enough, so that this host never reaches its threshold during the observation period, then the incremental delay with respect to the local job flow cannot be determined using formula (2.6). If this derivative is assumed to be 0, due to the lack of information, this will probably lead to unnecessary increase of the threshold value during the next iteration. Therefore, a heuristic modification must be made; namely, if a host does not hit its threshold during the observation interval, then it automatically decrements its threshold value by one.

We present numerical examples for a particular set of system parameters. However, we have observed similar results for a wide range of system parameters. We first consider a system with five host computers with the same utilization $u_1 = u_2 = u_3 = u_4 = u_5 = 0.5$. All the processors have the same job processing rate ($\mu_1 = ... = \mu_5 = 1.0$). The average communication delay of a job is assumed to be 10% of the mean job service time. The initial threshold values are $T_1^{init} = ... = T_5^{init} = 10$. Figure 5.1 shows the threshold value at host 1 of the simulated distributed system with respect to the number of iterations of the algorithm at that host. Each observation period consists of 50 busy cycles. We observed similar behavior of the algorithm for all the host computers of the system. It turns out that the algorithm converges very quickly to the optimal (or near-optimal) threshold value. Obviously, when the initial thresholds are large, it takes more iterations for the algorithm to converge and this is due to the fact that the optimal thresholds are usually small values [2]. Figure 5.2 shows the average response time of a job in the system of the five hosts as a function of time. The two curves correspond to different initial threshold values. The solid curve is derived for initial values $T_1^{init} = ... = T_5^{init} = 10$ while the dotted curve is derived for initial threshold values $T_1^{init} = ... = T_5^{init} = 15$. As we can expect the system with the larger initial threshold values converges slower. A comparison is made with a system with no load balancing at all (NLB), where all the hosts are modeled as M/M/1 systems with utilization 0.5.

In Figures 5.3 and 5.4 we consider experiments where host computers have different utilizations. In Figure 5.3 hosts 1, 2 and 3 have utilization 0.9 and hosts 4 and 5 have utilization 0.5. The average response time of a job is shown as a function of time. Two curves are presented corresponding to different initial threshold values and a comparison is made with a system with no load balancing. The same experiment is repeated for utilizations $u_1 = u_2 = u_3 = 1.2$ and $u_4 = u_5 = 0.5$ (Fig. 5.4). Obviously, hosts 1, 2 and 3 are saturated without load balancing. Since these three nodes are overloaded, a busy cycle termination occurs very rarely and so does a threshold update. Therefore, starting with large initial threshold values at the overloaded nodes results in larger average response time of a job (Fig. 5.4).

We next study the *adaptivity* of the threshold updating algorithm in a dynamically changing system environment. In such a case, when there is an imbalance in incremental delays due to a change in the system environment, we expect that the algorithm corrects the imbalance properly so that the system performance may be improved. Figures 5.5 and 5.6 consider an example where hosts in the system have the same processing power ($\mu_1 = ... = \mu_5 = 1.0$). Initially all the five hosts have the same utilization of 0.5. Right after the 2500-th time unit (simulation time) the utilization of hosts 1, 2 and 3 changes to 0.9 (Figure 5.5) and 1.2 (Figure 5.6) respectively. Right after the 5000-th time unit (simulation time) the utilization of hosts 1, 2 and 3 returns to 0.5. As it turns out from the simulation results, the algorithm adapts smoothly to a change of increasing or decreasing workload. After a short transient time the average response time converges to the value we had observed for the corresponding system in a static environment (Figures 5.2, 5.3 and 5.4). In Figure 5.6 the solid curve indicates the behavior of the algorithm in a system changing environment when the initial threshol values are 10 for all the nodes. The dotted curve represents its behavior in the same environment but in this case threshold value is kept constant $T = 2$ for all the nodes.

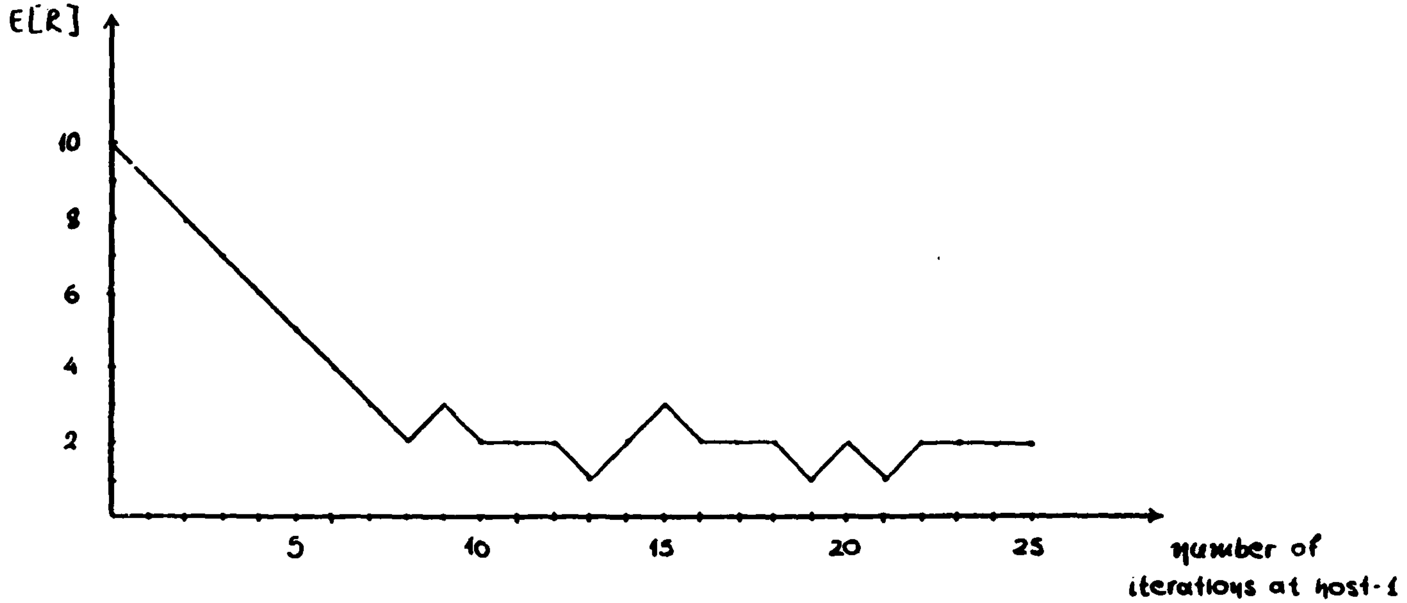


Fig. 5.1 - Behavior of the threshold updating algorithm at host-1 of a distributed system with 5 hosts ($T_1^{init} = ... = T_5^{init} = 10$ and $u_1 = ... = u_5 = 0.5$).

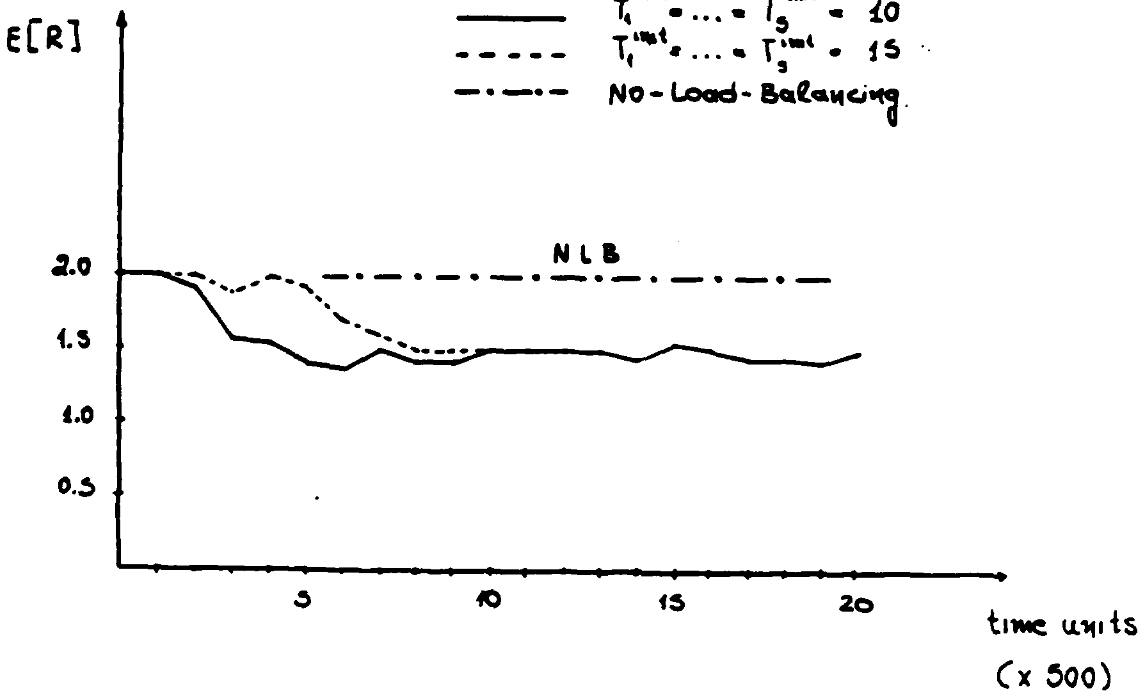


Fig. 5.2 - Average response time a job in the system of five hosts as a function of time ($u_1 = ... = u_5 = 0.5$).

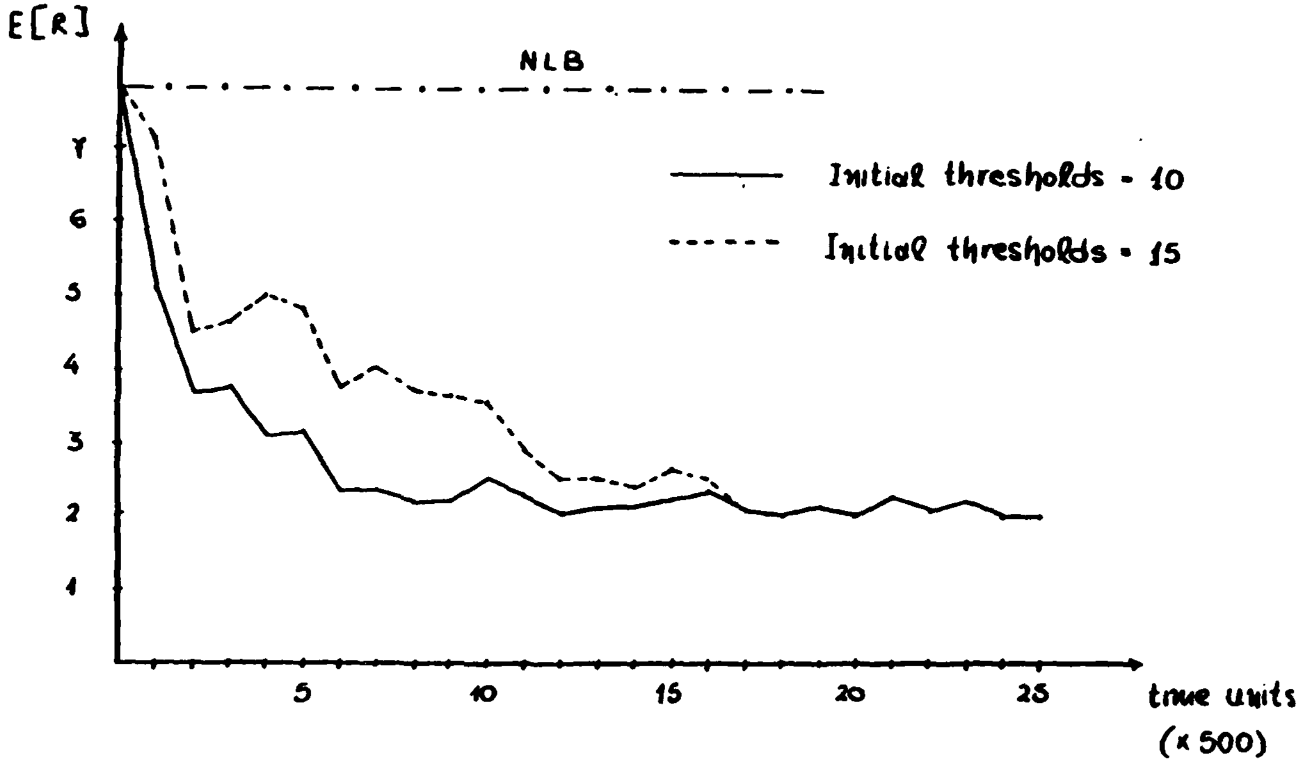


Fig. 5.3 - Average response time a job in the system of five hosts as a function of time ($u_1 = u_2 = u_3 = 0.9$ and $u_4 = u_5 = 0.5$).

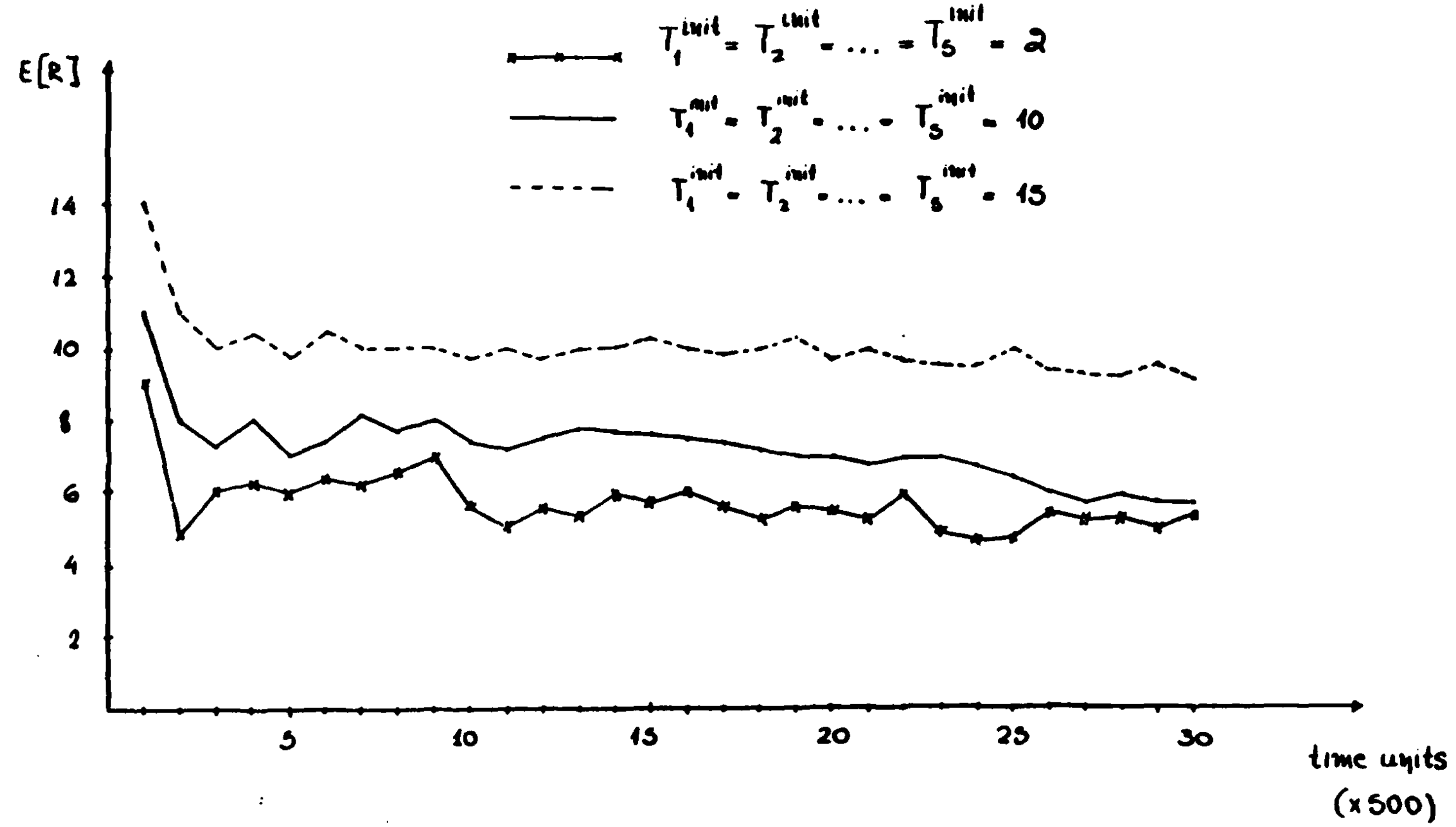


Fig. 5.4 - Average response time of a job in the system of five hosts as a function of time ($u_1 = u_2 = u_3 = 1.2$ and $u_4 = u_5 = 0.5$).

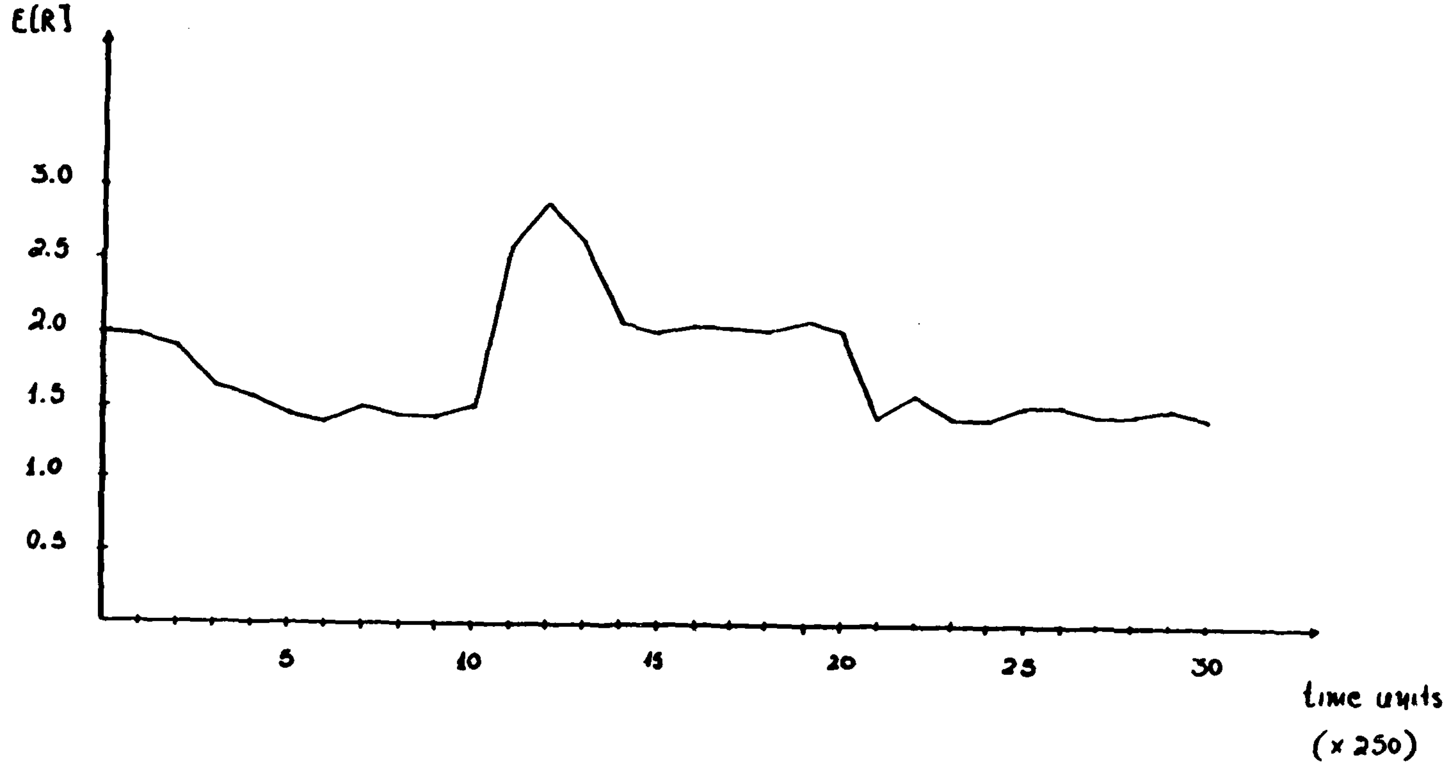


Fig. 5.5 - Behavior of the threshold updating algorithm in a changing system environment (utilization at nodes 1, 2 and 3 changes from 50% to 90% and back to 50%).

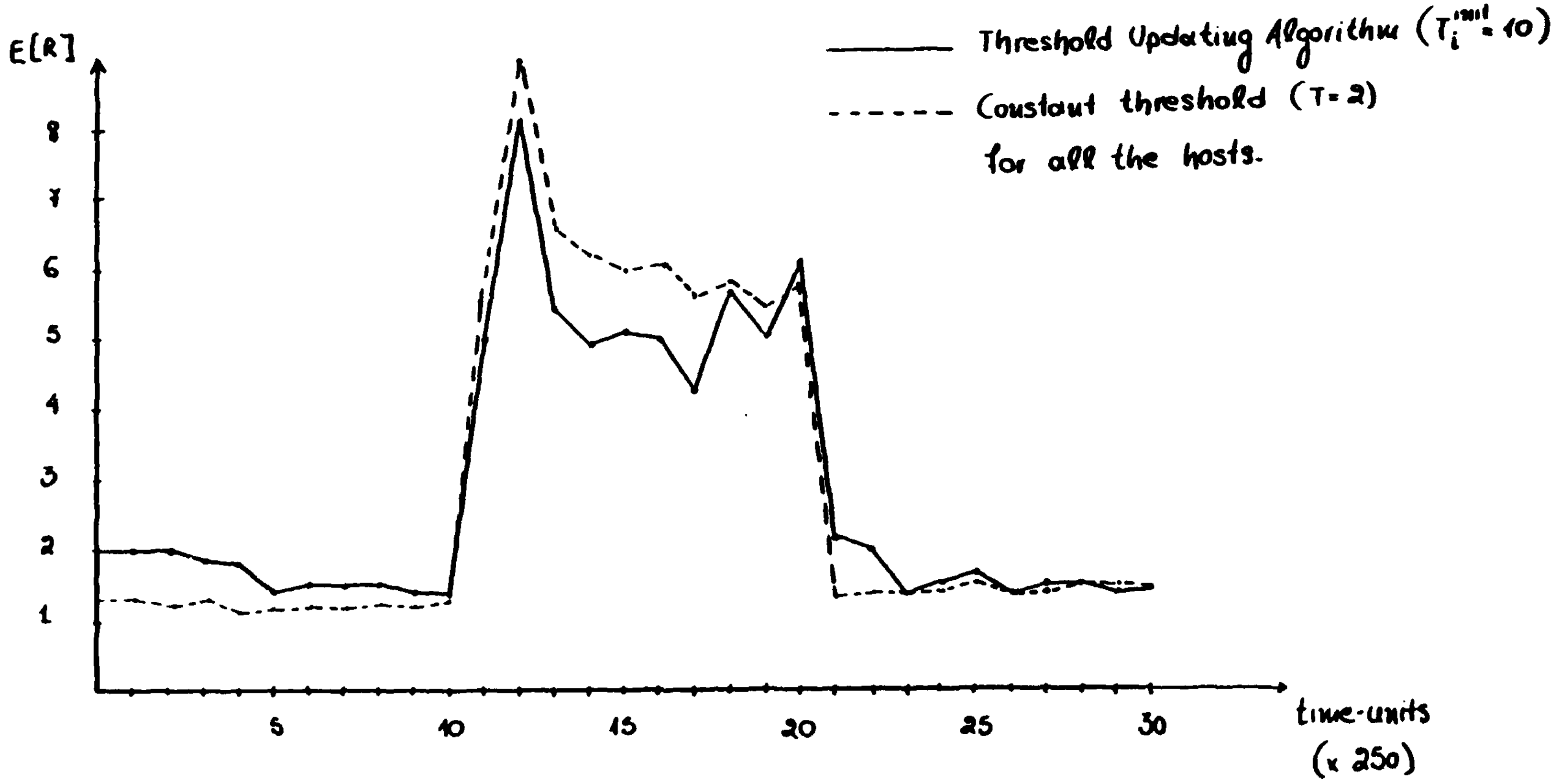


Fig. 5.6 - **Behavior of the threshold updating algorithm in a changing system environment (utilization at nodes 1, 2 and 3 changes from 50% to 120% and back to 50%).**

6. CONCLUSIONS

In this work we considered the problem of improving the performance of a distributed system by using *load balancing* techniques to smooth-out periods of high congestion at individual nodes. Specifically, we adopted a class of load balancing techniques, that use a threshold policy at each host to make decisions for job transfers. We put emphasis on the problem of determining the optimum values of threshold parameters in order to yield optimal or near-optimal performance. An efficient distributed algorithm has been adapted to the specific load balancing policy for that purpose. The algorithm is iterative in nature and threshold parameters are updated at each iteration. Between updates, each host computes and exchanges with other hosts gradient information. This information is then used to update load balancing parameters in the next iteration.

Obviously, the above distributed algorithm is based on the knowledge of performance sensitivity information. Each host must be able to estimate the change in throughput and expected queue length due to a change in either the threshold or the job arrival rate. Two different methods towards estimating the above gradients have been proposed in this work. Both of them effectively provide performance sensitivity estimates while the actual system is running (on-line estimation), by using real data gathered during an observation interval. In both the methods the arrival rate is estimated during the execution of the estimation algorithm.

Our approach towards estimating the gradients with respect to the threshold relies on the assumption that either the arrival process or the service time are exponentially distributed. The performance of the proposed estimation technique has been investigated through simulations. As it turns out from the simulation results, the estimation algorithm converges very fast even for short observation periods (i.e. 200 job completions). Furthermore, the memory requirements and the increase in running time are very low.

A different technique is proposed for estimating gradients with respect to the arrival rate. The method is extremely well suited to regenerative systems. In our case, the beginning of a new busy cycle is considered as the regeneration point. By evaluating the above estimation technique through simulations, we realized that the observation period must be long enough (e.g. 2000 busy cycles) in order for the estimator to give consistent estimates of the desired gradient. However, the method can be effectively applied to systems involved in the decentralized threshold scheduling policy discussed in this work, where we only need to compare derivatives without worrying about their absolute value.

Both the proposed estimation techniques have been imbedded in the updating threshold algorithm and simulation results have been produced for a system with five host computers modeled as single server queueing systems. It turns out that after a finite number of algorithm iterations, the behavior of the system, in a static environment is confined in a neighborhood of the optimal performance. A great improvement in the performance measure (average response time of a job) has been observed in the system executing the distributed load balancing algorithm compared to a system with no load balancing at all. Although we used short observation periods (50 or 100 busy cycles) in most of our experiments, the performance of the algorithm was not affected. It appears that accuracy in gradient estimation is not very crucial, since it is the *relative* values of the gradients, not their *absolute* values, which are more significant. Since the optimal thresholds are usually small values (e.g. less than six), it takes more iterations for the algorithm to converge if the initial thresholds are large. A heuristic modification has been made in order to improve the performance of the technique estimating gradients with respect to threshold, when the initial threshold values are large enough so that the nominal system does not even reach its threshold. It is interesting to notice that the overhead of exchanging gradients between the hosts of the distributed system is small since they are not an istantaneous information such as queue length. An interesting problem, is the *adaptivity* of the algorithm in a dynamically changing system environment. In such a case, when there is an imbalance in incremental delays due to a change in the system environment, we expect that the algorithm corrects the imbalance properly so that the system performance may be improved. Simulation results pointed out that the algorithm adapts smoothly to a change of the workload. When the system environment changes over time, this algorithm can run in the background so that it can track the system variation in a quasi-static manner [2].

REFERENCES

[1]. K. J. LEE, D. TOWSLEY, "Quasi-static decentralized load balancing with site constraints", submitted to ACM Sigmetrics Conf., 1987.

[2]. K. J. LEE, Load Balancing in Distributed Computer Systems, Ph.D. Thesis, Dept. Elect. & Comp. Eng., Univ. of Massachusetts, 1987.

[3]. K. J. LEE, D. TOWSLEY, "A comparison of priority-based decentralized load balancing policies", Proc. of Performance 1986 and ACM Sigmetrics Conf. 1986.

[4]. K. J. LEE, D. TOWSLEY, "Distributed Algorithms for decentralized load balancing in star-configured systems", submitted to *IEEE Trans. on Computers*, 1986.

[5]. D. EAGER, E. LAZOWSKA, J. ZAHORJAN, "Adaptive load sharing in homogenious distributed systems", *IEEE Trans. on Software Eng.*, SE-12, pp. 662-675, May 1986.

[6]. A. TANTAWI, D. TOWSLEY, "Optimal static load balancing in distributed computer systems", *Journal of ACM*, vol. 32, pp. 445-465, April 1985.

[7]. Y. WANG, R. MORRIS, "Load sharing in distributed systems", *IEEE Trans. on Computers*, vol. C-34, pp. 204-217, March 1985.

[8]. D. LUENBERGER, Introduction to Linear and Nonlinear Programming, Addison-Wesley, 1973.

[9]. C. CASSANDRAS, Y. HO, "An event domain formalism for sample path perturbation analysis of discrete event dynamic systems", *IEEE Trans. on Automatic Control*, AC-30, pp. 1217-1221, 1985.

[10]. R. SURI, M. ZAZANIS, "Perturbation Analysis gives strongly consistent sensitivity estimates for the M/G/1 queue", *Operations Research*, 1985.

[11]. C. CASSANDRAS, "On-line optimization of a flow control strategy", *IEEE Trans. on Automatic Control*, AC-31, Feb. 1986.

[12]. M. REIMAN, A. WEISS, "Sensitivity Analysis for Simulations via Likelihood Ratios", AT&T Bell Labs Report, 1987.

ANALYSIS OF FORK-JOIN JOBS USING PROCESSOR-SHARING[1]

C. G. Rommel[2], D. Towsley,[3] J. A. Stankovic[3]

University of Massachusetts
Amherst, MA 01003

Abstract

In this paper we derive an expression for the mean response time of a fork-join job in a single server processor-sharing queueing system. We also derive an expression for the mean response time of a fork-join job conditioned on the required service time of the largest task. In our approach a fork-join job is composed of a number of independent tasks which can be scheduled independently of each other. The job is considered complete once the last task completes. Each task service time is assumed to be an exponentially distributed random variable. We provide both lower and upper bounds on mean response time of fork-join jobs. The lower bound to the mean response time of the fork-join problem is very tight when the number of tasks in the job is large (· 7) and/or the server utilization is high. Finally, numerical results are developed that provide various insights such as that fact that processor-sharing scheduling at the job level is better than at the task level.

Key words: Fork-join, job scheduling, lower bound, performance evaluation, processor-sharing, queueing theory, upper bound, and task scheduling.

[1] This work was supported in part by the National Science Foundation under grant MCS-8104203 and by RADC under contract RI-44896X.

[2] Department of Electrical Engineering

[3] Dept. Computer and Information Sciences

1 Introduction

With the advent of programming languages that support parallel programming, (i.e., Concurrent Pascal [Han75], CSP[Hoa85], and Ada [Pyl81]) and multiprocessors, there is increasing interest in modeling the performance of parallel programs. In this paper, we evaluate the performance of a simple parallel program, a *fork-join* on a single server that uses processor-sharing as its scheduling policy. We derive an expression for the mean response time of a fork-join job and both lower and upper bounds on the mean response time. Our lower bound is very tight when the number tasks created by a fork-join program is large. In our problem a fork-join job is composed of a set of tasks each of which can be scheduled independently of the others. The job completes when the last task completes.

There are several reasons why this problem is of interest. First, up to now the literature dealing with the exact performance analysis of parallel programs has focussed on scheduling policies that serve tasks in a first-come first-serve ($FCFS$) manner. Consequently, this paper is a first step toward the analysis of parallel programs under other scheduling policies. We have chosen the processor-sharing discipline because of its usefulness in modeling round robin disciplines that are typically utilized in uniprocessors and multiprocessors.

There is a second practical reason motivating this work. In the future, a large fraction of software will be written using parallel processing constructs such as forks and joins. In most cases, the resulting programs will execute on multiprocessors. However, occasions will occur when the parallel program may be transported to a system containing a single processor. Consequently, it is important to understand the behavior of these parallel programs on a single processor. We will observe from our analysis that a fork-join job incurs a higher mean response time if the basic schedulable entity is a task than if the basic schedulable entity is the job. Consequently, care must be taken in moving parallel programs from multiprocessors to uniprocessors.

As stated before, the literature has not addressed the issue of fork-join jobs in a processor-sharing queueing system. However, processor-sharing and $FCFS$ fork-join job scheduling have been addressed separately.

Bulk arrivals with processor-sharing are examined in [KMR71]. In [KMR71] a general result is given for task response time conditioned on service attained when tasks arrive in bulks. Our approach is an extension of [KMR71] by considering the average job response time rather than the average task response time.

The behavior of fork-join jobs on both uniprocessors and multiprocessors using a *FCFS* scheduling policy has been evaluated by modeling the system as a bulk arrival queuing system [NTT87]. The behavior of fork-join jobs executing in a system where each task is processed on a dedicated processor has been studied in several papers [FH84], [NT85], [BM85], [TY86]. Some of this work has been extended to parallel programs characterized by arbitrary acyclic precedence graphs [BMT87]. In our approach we extend the uniprocessor work of [NTT87] from the *FCFS* to processor-sharing. In summary, our work may be viewed as an extension of aspects of both [KMR71] and [NTT87] to analyze fork-join jobs under processor-sharing when only one server is used.

The remainder of the paper is structured in the following manner. The model of the system is found in Section 2. The analysis leading to an expression for the mean response time of a job and the conditional response time of a job given the service time of the largest task is contained in Section 3. In Section 3 we also develop both lower and upper bounds for job response time. Section 4 compares various scheduling disciplines. A study of the behavior of the system as a function of different parameters is also given in Section 4. Some concluding remarks are found in Section 5.

2 Model

In our model a job is initially composed of a set of M tasks which arrive to a single server system according to a Poisson arrival process with parameter λ. Here the number of tasks, M, is considered to be a random variable with mean $a = E(M)$. Each task can be executed concurrently with tasks of the same or other jobs. Tasks are assumed to be scheduled independently of each other according to a processor-sharing queueing discipline. The service times of the tasks are assumed to be exponential random variables with mean $1/\mu$. The server utilization is $\rho = \lambda a/\mu$.

It is worth observing that if a job containing M tasks is scheduled as a single entity, its job service time is an M^{th} order Erlang random variable. We shall observe that the variation of processor-sharing where the task is the schedulable entity does not perform as well as the ones where the job is the schedulable entity.

3 Analysis

In this section we determine the average response time of a job conditioned on the service requirement of the longest task and the average response time of a job over all service times. Initially, we shall make no assumptions regarding the task service time distribution. We introduce the following notation:

- X - is the service time of a task with cumulative distribution $B(x) = P[X \leq x]$ and mean $1/\mu$,
- $F(x)$ - is the cumulative distribution of the remaining service time where $F(x) = 1 - B(x)$,
- $\hat{X}_i$ - the service time of the i^{th} largest task in a job, i.e., $\hat{X}_1 \geq \hat{X}_2 \geq \cdots \geq \hat{X}_M$,
- Y - is the service time of the largest task in a job, $Y = \hat{X}_1$,
- W - is the mean response time of a task in the job,
- t - is the mean response time of the job such that $t = E(\max_{1 \leq i \leq M}\{W_i\})$,
- $w(x)$ - is the mean response time of a task in the job conditioned on the service time, $X = x$, i.e., $w(x) = E(W|X = x)$.
- $t(x)$ - is the mean response time of the job conditioned on the service time of largest task, $Y = x$, i.e., $t(x) = E(T|Y = x)$,
- $M(x)$ - number of incomplete tasks in a job after each task has been entitled to x units of service, i.e., $M(x) = m \; iff \; \hat{X}_m \geq x > \hat{X}_{m+1}$, $1 \leq m \leq M$ and $\hat{X}_1 < x$,
- $t_m(x) = E(T|Y = x, M = m)$,
- $w_m(x) = E(W|Y = x, M = m)$,
- $w'(x) = dw(x)/dx$,
- $w'_m(x) = dw_m(x)/dx$,
- $t'(x) = dt(x)/dx$.

We next derive an integro-differential equation for the mean job response time, establish closed form expressions for $t_m(x)$ and t when X is an exponential random variable, and establish bounds for these quantities.

3.1 Derivation of the Processor-Sharing Integro-Differential Equation

Our derivation of the task-sharing integro-differential equation follows the development of [KMR71] and relies on the feedback approach to processor-sharing presented in [KC67]. Our system is composed of a single queue and a processor-sharing server. Jobs enter at the tail of the queue and are split into tasks. Figure 1 gives a diagram of the queueing system when a particular job arrives. In the analysis to follow we will examine the progress of this job through the system. This job is referred to as the *tagged job.*

Each task receives a quanta of service when it reaches the server. If the attained service of the task is less than the required service, the task is placed at the end of the queue. Otherwise, the task exits. In this approach we first analyze the response time of a fork-join job when the scheduler is a round-robin discipline with a time slice, q. Then we consider the limiting case of round-robin as the time slice quantum approaches zero.

First, we will define the following in the context of the round-robin discipline :

- n_i - the mean number of tasks in the system when the largest task of the job receives it's i^{th} quantum of service.
- $N(x)$ - the task density of the system given x units of attained service.
- t_i - is the average delay between the $(i \quad 1)^{th}$ quantum of service and the i^{th} quantum of service for the largest task including the actual quantum of service,
- σ_i - the probability that a task which has received iq units of service will require more than $(i+1)q$ seconds of service, and
- ς_i - the probability that a task of a job which has received iq seconds of service will require more than $(i+1)q$ seconds of service given that the largest task still requires additional service.

The average delay until the largest task of the job receives its first quantum of service is simply

$$t_1 = \sum_{j=0}^{\infty} n_j q + (m-1)q + q. \tag{1}$$

We assume here that the largest task enters at the end of the queue. This ordering is not important since we consider the limit as $q \to 0$. Because of this limit, any existing tasks share the processor immediately. The first term in the above equation represents the delay due to tasks ahead of the newly arriving job. The second term models the effect of the delay due the members of the newly arriving job in front of the largest task. The last term gives the delay due to the first quantum of service of the largest task.

By induction, the mean time for the largest task to receive its i^{th} quantum of service is given as the sum of four terms.

$$\begin{aligned} t_i = & \sum_{j=0}^{\infty} n_j \sigma_j \sigma_{j+1} .. \sigma_{j+i-2} q + \\ & \sum_{j=1}^{i-1} \lambda a t_j \sigma_0 \sigma_1 .. \sigma_{i-j-2} q + \\ & (m-1) q \prod_{j=0}^{i-2} \varsigma_j + q . \end{aligned} \tag{2}$$

The first term is associated with tasks found when the job arrived at the site; the next term is associated with tasks which arrived after the job; the third term is due to the tasks of the job which remain after receiving i quanta of service; and q is the quantum of service that the largest member receives. Following Kleinrock's development we divide by q leaving :

$$\begin{aligned} t_i/q = & \sum_{j=0}^{\infty} n_j \sigma_j \sigma_{j+1} .. \sigma_{j+i-2} + \\ & \sum_{j=1}^{i-1} \lambda a t_j \sigma_0 \sigma_1 .. \sigma_{i-j-2} + \\ & (m-1) \prod_{j=0}^{i-2} \varsigma_j + 1. \end{aligned} \tag{3}$$

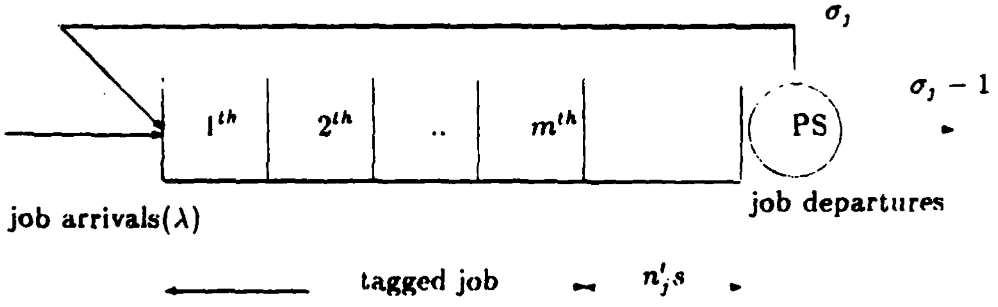


Figure 1: Process-Sharing Diagram after the Arrival of a Tagged Job

In the limit as $i \to \infty$, $j \to \infty$ such that

$iq \to x$, $jq \to y$, and $q \to 0$, we obtain the following limits:

$$t_i/q \to t'_m(x), \tag{4}$$

$$t_j \to t'_m(y)dy, \tag{5}$$

$$n_i \to N(y)dy, \tag{6}$$

$$\sigma_j\sigma_{j-i+1}..\sigma_{j+i-2} \to \frac{1 - B(y+x)}{1 - B(y)}, \tag{7}$$

$$\sigma_0\sigma_1..\sigma_{i-j-2} \to 1 - B(x-y), \tag{8}$$

$$1 + (m-1)\prod_{j=0}^{i-2} \varsigma_j \to E(M(x)|Y > x). \tag{9}$$

$$\tag{10}$$

By taking the limits of both sides of equation (3) as $q \to 0$, we obtain

$$\begin{aligned} t'_m(x) = {} & \int_0^\infty N(y)\frac{(1-B(y+x))}{(1-B(y))}dy + \\ & \lambda a\int_0^y t'_m(y)(1-B(x-y))dy + E(M(x)|Y > x). \end{aligned} \tag{11}$$

By noting that the average density of tasks [KC67] is

$$N(y) = \lambda a[1 - B(y)]w'(y) \tag{12}$$

we obtain our first main result,

$$\begin{aligned} t'_m(x) = {} & \lambda a\int_0^\infty w'_m(y)(1-B(y+x))dy + \\ & \lambda a\int_0^x t'_m(y)(1-B(x-y))dy \\ & + E(M(x)|Y > x). \end{aligned} \tag{13}$$

We remind the reader that the above equation makes no assumption regarding the nature of the service time distribution.

3.2 Fork-Join Jobs with Exponential Service Times and Processor-Sharing

If we assume exponential service times with mean $1/\mu$ for each task, then several simplifications occur. First, since the cumulative distribution is now exponential, we have

$$(1 - B(y + x)) = e^{-\mu(y+x)}, \tag{14}$$

$$(1 - B(x - y)) = e^{-\mu(x-y)}. \tag{15}$$

From the results of [KMR71] we know that

$$w'_m(y) = \frac{1}{(1-\rho)} + \frac{(a-1)(2-\rho)e^{-\mu(1-\rho)y}}{2(1-\rho)}. \tag{16}$$

Then by integration we obtain

$$\lambda a \int_0^{\infty} w'_m(y)(1 - B(y + x))dy = \frac{0.5\rho(a+1)e^{-\mu x}}{(1-\rho)}. \tag{17}$$

Using the exponential assumption we now have

$$\lambda a \int_0^x t'_m(y)(1 - B(x - y))dy = \rho\mu \int_0^x t'_m(y)e^{-\mu(x-y)}dy. \tag{18}$$

Let us now focus on $E(M(x)|Y > x)$. By using the definition of the expectation of a random variable we have

$$E(M(x)|Y > x) = \sum_{k=1}^{m} kP(M(x) = k|Y > x).. \tag{19}$$

From the definition of conditional probability we know that

$$P(M(x) = k|Y > x) = \frac{P(M(x) = k \, \text{and} \, Y > x)}{P(Y > x)}. \tag{20}$$

Since the event $(M(x) = k) \subseteq$ event $(Y < x)$ $k \geq 1$, we have

$$P(M(x) = k \text{ and } Y < x) = \begin{cases} 0 & , k = 0 \\ P(M(x) = k) & , 0 < k \leq m. \end{cases} \tag{21}$$

By observing that $M(x)$ is a binomial random variable, we have

$$P(k) = \binom{m}{k}[1 - F(x)]^{m-k}[F(x)]^{k}, k \geq 1. \tag{22}$$

By direct substitution we have

$$E(M(x)|Y < x) = \frac{\sum_{k=1}^{m} k\binom{m}{k}[1 - F(x)]^{m-k}[F(x)]^{k}}{P(Y < x)}. \tag{23}$$

If we now consider the denominator of the above expression, we see that by the basic probability axiom and substitution we have

$$P(Y < x) = 1 - P(Y > x) = 1 - (1 - e^{-\mu x})^{m}. \tag{24}$$

Then noting that the numerator is the expectation of a binomial random variable with $q = [1 - F(x)]$ and $p = F(x)$ and that $F(x) = e^{-\mu x}$, we have our desired result:

$$E(M(x)|Y < x) = \frac{me^{-\mu x}}{1 - (1 - e^{-\mu x})^{m}}. \tag{25}$$

When we combine these equations together, we form the integro-differential equation for the job response time for exponential service requirements.

$$\begin{aligned} t'_m(x)) = {} & \frac{0.5\rho(a + 1)}{(1 - \rho)} e^{-\mu x} + \\ & e^{-\mu x}\rho\mu\int_0^x t'_m(y)e^{\mu y}dy + \\ & \frac{me^{-\mu x}}{1 - (1 - e^{-\mu x})^{m}}. \end{aligned} \tag{26}$$

Equation (26) is our second important result.

To eliminate the exponential product in the integral of equation (26) we make the following substitution

$$t'_m(x) = R'(x)e^{-\mu x}. \tag{27}$$

By some algebra we may formulate the above equations into the following linear differential equation,

$$R'(x) - \rho\mu R(x) = -\rho\mu R(0) + \frac{0.5\rho}{(1-\rho)}(a+1) + \frac{m}{(1-(1-e^{-\mu x})^m)}. \tag{28}$$

Equation (28) may be solved either analytically or numerically by standard techniques. To determine the average response time of the job response time, $t_m(x)$ we integrate $R'(x)e^{-\mu x}$. Therefore,

$$t_m(x) = \int_0^x R'(y)e^{-\mu y}dy. \tag{29}$$

We now solve for the average response time of the job by conditioning on Y. The distribution of Y for the largest task is the same as the maximum order statistic of the exponential distribution. It is given as

$$f_{Y|M}(x|m) = m\mu(1-e^{-\mu x})^{m-1}e^{-\mu x}. \tag{30}$$

Then the average job response time, t is simply

$$t = \int_0^\infty t_m(y)f_{Y|M}(y|m)dy. \tag{31}$$

This completes our general analysis for the response time of a fork-join job using processor-sharing scheduling with exponential service requirements.

3.3 Closed Form Solution to the Fork-Join Job Problem

There are several ways to solve for equation (28). One way is to use numerical methods to determine R, $t'_m(x)$, and $t_m(x)$. We use a numerical approach in the section 4 because of its simplicity. However, to define the contributions of various parameters to the response time of the job, we derive a closed form solution to equation (28) in this section.

To determine a closed form expression for the response time, we expand $m/(1-(1-e^{-\mu x})^m)$ in terms of a sum of exponentials such that

$$m/(1-(1-e^{-\mu x})^m) = e^{\mu x} + \sum_{j=0}^{\infty} c_{mj} e^{-j\mu x}. \tag{32}$$

Appendix A gives a development for equation (32) and describes how the parameters, $c_{mj}, j = 0, 1, ..\infty$ may be obtained. This development is based on a partial fraction expansion approach in order to avoid instability problems. Next, we observe that equation (28) is a simple first order differential equation. The homogeneous solution is

$$R_h(x) = Ce^{\rho x}, \tag{33}$$

where C is a integration constant. To find the particular solution we use the expansion of $m/(1-(1-e^{-\mu x})^m)$ given by equation (32).

The total expansion of the right hand side of equation (28) is

$$K + c_{m0} + e^{\mu x} + \sum_{j=1}^{\infty} c_{mi} e^{-j\mu x} \tag{34}$$

where K is the constant term from equation (28) and c_{m_i} are the constants of the expansion of $m/(1-(1-e^{-\mu x})^m)$. The particular solution is given by

$$R_p(x) = \frac{K + c_{m0}}{-\rho\mu} + \frac{e^{\mu x}}{\mu(1-\rho)} - \sum_{j=1}^{\infty} \frac{c_{mj} e^{-j\mu x}}{\mu(j+\rho)} \tag{35}$$

and the complete solution is then

$$R(x) = Ce^{\rho x} + \frac{K + c_{m0}}{\rho\mu} + \frac{e^{\mu x}}{\mu(1-\rho)} - \sum_{j=1}^{\infty} \frac{c_{mj} e^{-j\mu x}}{\mu(j+\rho)} . \tag{36}$$

By using the procedures in the previous section, we obtain the solution for the response time conditioned on the service requirement

$$t_m(x) = \frac{x}{1-\rho} + \frac{\rho C}{(1-\rho)}(1 - e^{-(1-\rho)\mu x}) + \frac{1}{\mu}\sum_{j=1}^{\infty} \frac{jc_{mj}(1 - e^{-(j+1)\mu x})}{(j+1)(j+\rho)} \tag{37}$$

with the initial condition that $t_m(0) = 0$. The factor ρC may be determined from the initial conditions on the equation (26) and the derivative of $t_m(x)$. The result is

$$\mu\rho C = \frac{1}{1-\rho}(0.5\rho(a+1) - 1) + m - \sum_{j=1}^{\infty} \frac{jc_{mj}}{(j+\rho)} . \tag{38}$$

We can also determine the average response time of the job over all service times. This result is given as

$$t = \frac{1}{\mu}\Big[\frac{H_m}{1-\rho} + \frac{\rho C}{(1-\rho)}\Big(1 - \frac{m!}{\prod_{j=1}^{m}(1+j-\rho)}\Big) + \sum_{j=1}^{\infty} \frac{jc_{mj}}{(j+1)(j+\rho)}\Big(1 - \frac{m!}{\prod_{i=1}^{m}(1+j+i)}\Big)\Big] \tag{39}$$

where H_m is the harmonic series($H_m = \sum_{i=1}^{m} 1/m$). Equation (39) is valid for $\rho > 0$. An expression can be developed for $\rho = 0$ yielding

$$t = \frac{m}{\mu} . \tag{40}$$

Before continuing, we investigate the term

$$\frac{1}{(1-\rho)}\Big(1 - \frac{m!}{\prod_{j=1}^{m}(1+j-\rho)}\Big). \tag{41}$$

in equation (39).

We observe that it can be expressed as

$$\frac{\sum_{j=0}^{m-1} d_j \rho^j}{\prod_{j=1}^{m}(1+j-\rho)}) \tag{42}$$

where the d_j terms are the result of synthetic division by $(1 - \rho)$ into $(1 - \frac{m!}{\prod_{j=1}^{m}(1+j-\rho)})$. This implies that our result depends on $1/(1 - \rho)$ and not $1/(1 - \rho)^2$.

We also observe that equation (42) converges to H_m as ρ approaches one. With these facts in mind we may make the following observations:

1. the mean response time of a job grows in a with slope $\frac{1}{\mu}\frac{0.5\rho}{(1-\rho)^2}(1 - \frac{m!}{\prod_{j=1}^{m}(1+j-\rho)})$ and
2. the mean response time of a job grows nonlinearly in m as a function of ρ.

3.4 Bounds on Job Mean Response Time

In this section we develop bounds on the average response time of fork-join jobs. These bounds are important since the evaluation of the exact mean response time becomes computationally complex as m grows.

We obtain a lower bound, $R^{lb}(x)$, by bounding the $m/(1 - (1 - e^{-\mu x})^m)$ term of the right hand side of equation (28) by m. The proof of this may be found in [Rom87]. This bound becomes tight when either $m \to \infty$, or $\mu^- \to 0$. The resulting differential equation

$$t_m^{lb'}(x)) = \frac{0.5\rho(a+1)}{(1-\rho)}e^{-\mu x} + e^{-\mu x}\rho\mu\int_0^x t'_m(y)e^{\mu y}dy + m \qquad (43)$$

can be easily solved using the methods of section 3.3 to yield

$$t_m^{lb}(x) = \rho C\frac{1 - e^{-(1-\rho)\mu x}}{(1-\rho)}. \qquad (44)$$

Removal of the conditioning on $Y = x$ yields

$$t^{lb} = \frac{1}{\mu}\left[\frac{0.5\rho(a+1)}{(1-\rho)^2} + \frac{m}{(1-\rho)}\left[1 - \frac{m!}{\prod_{j=1}^{m}(1+j-\rho)}\right]\right]. \qquad (45)$$

Equation (45) gives similar insights into processor-sharing job response time as does equation(39). We see similar dependence of the response time on a, ρ, and m.

An upper bound may be derived in a similar fashion to the lower bound. The bound itself results from bounding the $me^{-\mu x}/(1-(1-e^{-\mu x})^m)$ term of the right hand side of equation (26) by m. This upper bound becomes tight when either the task size, $m \to 1$, or $\mu \to 0$. The results for the conditioned mean response time and unconditional mean response time are given without proof :

$$t_m^{ub}(x) = \rho C \frac{1 - e^{-(1-\rho)\mu x}}{(1-\rho)} + \frac{mx}{1-\rho} \tag{46}$$

where

$$\rho\mu C = \frac{1}{1-\rho}(0.5\rho(a+1)) - m\frac{\rho}{1-\rho}; \tag{47}$$

and

$$t^{ub} = \frac{1}{\mu}\left[\frac{mH_m}{(1-\rho)} + \left(\frac{0.5\rho(a+1)}{(1-\rho)^2} - \frac{\rho m}{(1-\rho)^2}\right)\left(1 - \frac{m!}{\prod_{j=1}^{m}(1+j-\rho)}\right)\right]. \tag{48}$$

This completes our treatment of the job response time.

4 Numerical Results

In this section we give numerical results for the scheduling algorithm we have just analyzed - the processor-sharing(PS) discipline in which the task is the basic schedulable entity. The average response time of the job for this algorithm is given by equation (31). We refer to this model as the $M/G/1-PS-TASK$ model. Our analysis will be based on a numerical solutions of equation(31). In section 4.1 we define the details of the solution technique in solving equation (31). Section 4.2 compares the average response time of two algorithms: task scheduling using processor-sharing to job scheduling using processor-sharing. Section 4.3 provides a comparison of task scheduling with processor-sharing to job scheduling using $FCFS$. Section 4.4 examines the lower bound given in equation (45) and the upper bound given by equation(48). Finally, Section 4.5 describes results for task scheduling under processor-sharing in which average number tasks per job at the server is not equal to the number of tasks in a particular job, i.e. $a \neq m$.

4.1 Numerical Details

To investigate the issues in sections 4.2 through 4.5 inclusive we examine response time given by equation (28) for jobs from 1 to 8 tasks under utilizations from 0.1 to 0.9 with a average service time of 10000 milliseconds. We obtain the mean response time of the $M/G/1-PS-TASK$ system using a fourth order Runge-Kutta method for solving equation (31). We use a increment size of 200 milliseconds and 4000 points. When we consider processor-sharing with job scheduling, $FCFS$ with job scheduling, and average task response time under $M/G/1-PS-TASK$, we also assume that all jobs consist of exactly the same number of tasks, *i.e.* $a=m$. This is done for ease of comparison only. In section 4.5 we consider the case when $a \neq m$ using the same parameters above with a varying independently of m.

4.2 Task Scheduling and Job Scheduling under Processor-Sharing

The first scheduling algorithm, which we compare to the $M/G/1-PS-TASK$, is the PS discipline where the job is the smallest schedulable entity. The fact that the job might be composed of tasks is immaterial to the scheduler. We refer to this as the $M/G/1-PS-JOB$ model. The response time of the job is given by

$$t = \frac{m}{\mu}\left(\frac{1}{1-\rho}\right). \tag{49}$$

Figure 2 plots the ratio of the average response time of the $M/G/1-PS-TASK$ system scheduling to the $M/G/1-PS-JOB$ system for jobs of 1, 3, 5, and 7 tasks. Several observations can be made from our data. First, we see that as the utilization, ρ, approaches zero the performance of the $M/G/1-PS-TASK$ system approaches the performance of the $M/G/1-PS-JOB$ system independent of the number of tasks in a job. This is expected since no queueing occurs in both conditions. In addition, the curve representing a job with only 1 task ($m=1$) in Figure 2 shows that the ratio of the $M/G/1-PS-JOB$ system to the $M/G/1-PS-TASK$ system equals 1 independent of the number of tasks. In general, we conclude that the performance of the two systems approach each other as $m \to 1$. This result is also expected. However, as the number of tasks in a job increases and the utilization increases, then the $M/G/1-PS-JOB$ system gives better performance than the $M/G/1-PS-TASK$ system. This leads to an important observation that *processor-sharing scheduling at the job level is better than that at the task level.* This is can be explained since the amount of service given to a job in the $M/G/1 - PS - TASK$ system is a linear function of the number of tasks. Consequently, large jobs get preferential treatment. As an example, when $m = 7$ and $\rho = .9$, we see the job response time is nearly 50% higher for the $M/G/1 - PS - TASK$ than the $M/G/1 - PS - JOB$ system. Moreover, we see an increase in the ratio of response times with the utilization, ρ. This is also expected since as ρ increases small tasks must share the processor with a larger number of tasks over a longer period of time. Thus, a job composed of small tasks are delayed longer in the task scheduling approach than the job scheduling approach.

4.3 Task Scheduling under Processor-Sharing and Job Scheduling under $FCFS$

The second numerical study we perform is to compare task scheduling under processor-sharing to job scheduling under $FCFS$. Although the tasks are scheduled sequentially, each task service time is exponential. In this model the service time is Erlangian. We refer to this as the $M/G/1 - FCFS$ model. The expected delay is given in [All78] as

$$t = \frac{m}{\mu}\left(1 + \rho\frac{0.5(1 + 1/m)}{1 - \rho}\right). \qquad (50)$$

Figure 3 displays the ratio between the job response time under task scheduling under processor-sharing to the job response time under $FCFS$. We see that even for low utilization, the $M/G/1 - FCFS$ system performs better than processor-sharing approach. Only when the utilization approaches zero do both algorithms have the same performance. We see that ratios are greater than for the simple task and job scheduling algorithms. This is expected since the coefficient of variation of the service time of the $M/G/1 - FCFS$ system is less than the corresponding $M/G/1 - PS - JOB$ system. Therefore, it is expected to be less than the $M/G/1 - PS - TASK$ system. To continue the example from section 4.2, we see that at $m = 7$ and $\rho = .9$ the job response time is nearly 200% more for the $M/G/1 - PS - TASK$ than the $M/G/1 - FCFS$ system.

4.4 Lower and Upper Bounds on Job Response Time

Now we want to compare our bounds to the exact solution. Figure 4a shows the tightness of the lower bound equation (45) developed in the previous section. This figure gives the difference between the lower bound and the exact solutions as a percentage of the latter. We give curves for the task number, m, of 1, 3, 5, and 7. We see that the lower bound approaches the exact solution as $\rho \to 1$. We also observe that the bound as $\rho \to 1$ is independent of the task count. This is expected since the dominant factor is the utilization of the server. In addition, we observe that for tasks counts of 5 or greater our bound is within 15% of the exact solution. For task counts of 7, our bound is nearly within 10% of the exact solution where the utilization is low ($\rho < 0.7$) and within 5% for high utilization ($\rho > 0.7$). It appears that the bound is close enough for $m \geq 7$ to be used as an approximate solution. Figures 4b, 4c and 4d plot the response time of the job using the lower bound, the exact solution and the upper bound. In 4b we use $m = 2$. In 4c we use $m = 5$. Finally, in 4d we use $m = 7$. We observe that both bounds are good for $m = 2$. However, the upper bound becomes worst as we increase m whereas the lower bound becomes tighter as we increase m. As an example observe the difference between the lower bound and exact solutions for $m = 7$ and $\rho = 0.8$. There is clearly little difference.

4.5 Job Response Time with $a \neq m$

The next experiment considers the effect of the overall average number of tasks, a, on the response time of the tagged job. The reader should recall from section 3.1 that a tagged job is simply a particular job with a given m. The number of tasks in this job may be different from the average number of tasks for all jobs at a server since a need not equal m. In this experiment we consider the number of tasks of the tagged job to take on values of 1, 3, 5, and 7 tasks per job while varying the value of the overall average number of tasks per job to 1 to 7. Figures 5a, 5b, 5c, and 5d give the response times in milliseconds for values of $\rho = 0.1, 0.5, 0.7$ and 0.9 with a as the independent parameter. As observed in Section 3.3, we see that a plays linear role in response time for a give value of ρ. In fact, we can determine the slope of the response time curve from equation(39). It is given as $\frac{1}{\mu}\frac{0.5\rho}{(1-\rho)^2}\left(1 - \frac{m}{\prod_{j=1}^{m}(1+j\rho)}\right)$.

Figures 6a, 6b, 6c and 6d plot the response time as a function of m instead of a. We see a clear nonlinear dependence on m as expected from equation(39). If we observe

Figure 6d, we see the most nonlinear dependence. However, after about $\rho = 0.5$, we observe that the increase in response time is approximately a linear function of m. This is expected since our lower bound becomes both tight and linear as $\rho \to 1$.

5 Conclusions

In summary, we have developed an exact solution for solving the fork-join processor-sharing with a single server queue. We have a solution for the rate of the response time conditioned on service time requirements. We also have a numerical solution for the response time for fork-join jobs with exponential service times. We have developed both lower and upper bounds for the fork-join processor-sharing model.

Our results demonstrate that scheduling fork-join jobs under processor-sharing should be done at the job level and not at the task level. This consideration is clearly important for single server applications. It is also important for multi-server systems applications when these systems degrade to single server systems. In these applications, jobs may be split for scheduling at different servers. However, the tasks of a job may be forced to remain together at a particular site due to possible faults in a communication network, network scheduling problems, or remote server overloading. From our observations, in such circumstances the job should not be split at the site. In fact, if the tasks of the job are grouped together, the overall system performance is improved. In conclusion, fork-join jobs should be carefully scheduled when moved from a multiprocessor system to a single processor system when processor-sharing is used.

Acknowledgments: We would liked to acknowledge Professor Gerald Giessert's assistance is obtaining equation (25).

Appendix A Exponential Series For Equation(25)

We consider a special approach to the expansion of equation(25) to prevent stability problems which arise when using the standard geometric expansion. Our expansion begins by noting that

$$\frac{m}{(1 \quad (1 \quad e^{\quad \mu x})^{m})} \qquad \frac{m}{z^{m} \quad 1} \tag{51}$$

$$where\, z \quad 1 - e^{\quad \mu x}. \tag{52}$$

We also know that

$$\frac{m}{(1 \quad (1 \quad e^{\quad \mu x})^{m})} \qquad m \prod_{j \quad 0}^{m} \frac{1}{z \quad r_j} \tag{53}$$

$$where\, r_j \quad e^{2\pi j i/m}\, j \quad 0,1,2,\ldots,m \quad 1\, and \tag{54}$$

$$i \quad \sqrt{\quad 1}. \tag{55}$$

By partial fraction expansion and substitution we have

$$\frac{m}{(1 \quad (1 \quad e^{\quad \mu x})^{m})} \qquad e^{\mu x} + \sum_{j \quad 1}^{m} \frac{g_j}{1 \quad r_j \quad e^{\quad \mu x}} \tag{56}$$

$$where\, g_j \qquad m \prod_{k \quad 0, k \neq j}^{m} \frac{1}{z \quad r_k} \Big|_{z \quad r_j}. \tag{57}$$

The values for g_j are complex numbers in general. To expand the above equation in terms of exponentials we formulate the term associated with the j^{th} root term as

$$\frac{g_j/(1 + b_j \quad r_j)}{1 \quad (b_j + e^{\quad \mu x})/(1 + b_j \quad r_j)}. \tag{58}$$

The value of b_j must be inserted so that the following series expansion converges as a geometric series. It can be shown that convergence occurs when $cos(2\pi j/m)$ $0.5/(b_j + 1)$. We may now find the c_{mj} terms. To do this now formulate algorithm. *Algorithm* 1.

Algorithm 1: Determination of Coefficients, c_{ij}

1. Initialize all coefficients. Set $\forall c_{mj} = 0$.
2. Select the first root. Let $j = 1$.
3. Determine the proper value of b_j. Let $b_j = -1$.
4. Let $b_j = b_j + 1$. If $cos(2\pi j/m) < 0.5/(b_j + 1)$ then *Step*4 else *Step*5.
5. Modify the coefficients. First, let $k = 0$.
6. For $l = 0$ to k let $c_{ml} = c_{mk} + g_j\binom{k}{l}b_j^{k-l}/(b_j + 1 - r_j)^{k+1}$.
7. If the absolute value of $1/(b_j + 1 - r_j)^k$ is less than an acceptable error then proceed to *Step*8 else let $k = k + 1$ and proceed to *Step*6.
8. Select the next root. Let $j = j + 1$. If $j > (m - 1)$ then *Stop* else goto *Step*3.

For values of $m < 6$ the value of b is zero. Thus, the algorithm is simplified under this constraint since the sum in step 6 invokes only the term when $k = l$.

References

[All78] Arnold Allen. *Probability, Statistics and Queueing Theory.* Academic Press, New York, New York, 1978.

[BM85] F. Baccelli and A. Makowski. Simple computable bounds for the fork-join queue. *Proc. Conf. Inform. Sci. Systems*, 1985.

[BMT87] F. Baccelli, W. Massey, and D. Towsley. Acyclic fork-join queueing networks. *submitted to JACM*, 1987.

[FH84] L. Flatto and S Hahn. Two parallel queues created by arrivals with two demands. *Siam J. Appl. Math.*, 44, 1984.

[Han75] P. Brinch Hansen. The programming language concurrent pascal. *IEEE Transaction on Software Engineering.*, 1. 1975.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[KC67] Leonard Kleinrock and E.G. Coffman. Distribution of attained service in time-shared systems. *Journal of Computer System Science*, 1, 1967.

[KMR71] Leonard Kleinrock, R. Muntz, and Rodemich. The processor-sharing queueing models for time-shared systems with bulk arrivals. *Networks*, 1, 1971.

[NT85] R. Nelson and A.N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IBM Report RC11481*, 1985. to appear in IEEE Transactions on Computers.

[NTT87] R. Nelson, D. Towsley, and A. Tantawi. Performance analysis of parallel processing systems. *to be presented at SIGMETRICS '87*, 1987.

[Pyl81] I.C. Pyle. *The Ada Programming Language*. Prentice-Hall International, London, 1981.

[Rom87] C. Gary Rommel. *Distributed Programs*. PhD thesis, University of Massachusetts, 1987. in preparation.

[TY86] D. Towsley and S. P. Yu. Bounds for two server fork-join queueing systems. *submitted to Operations Research*, 1986.

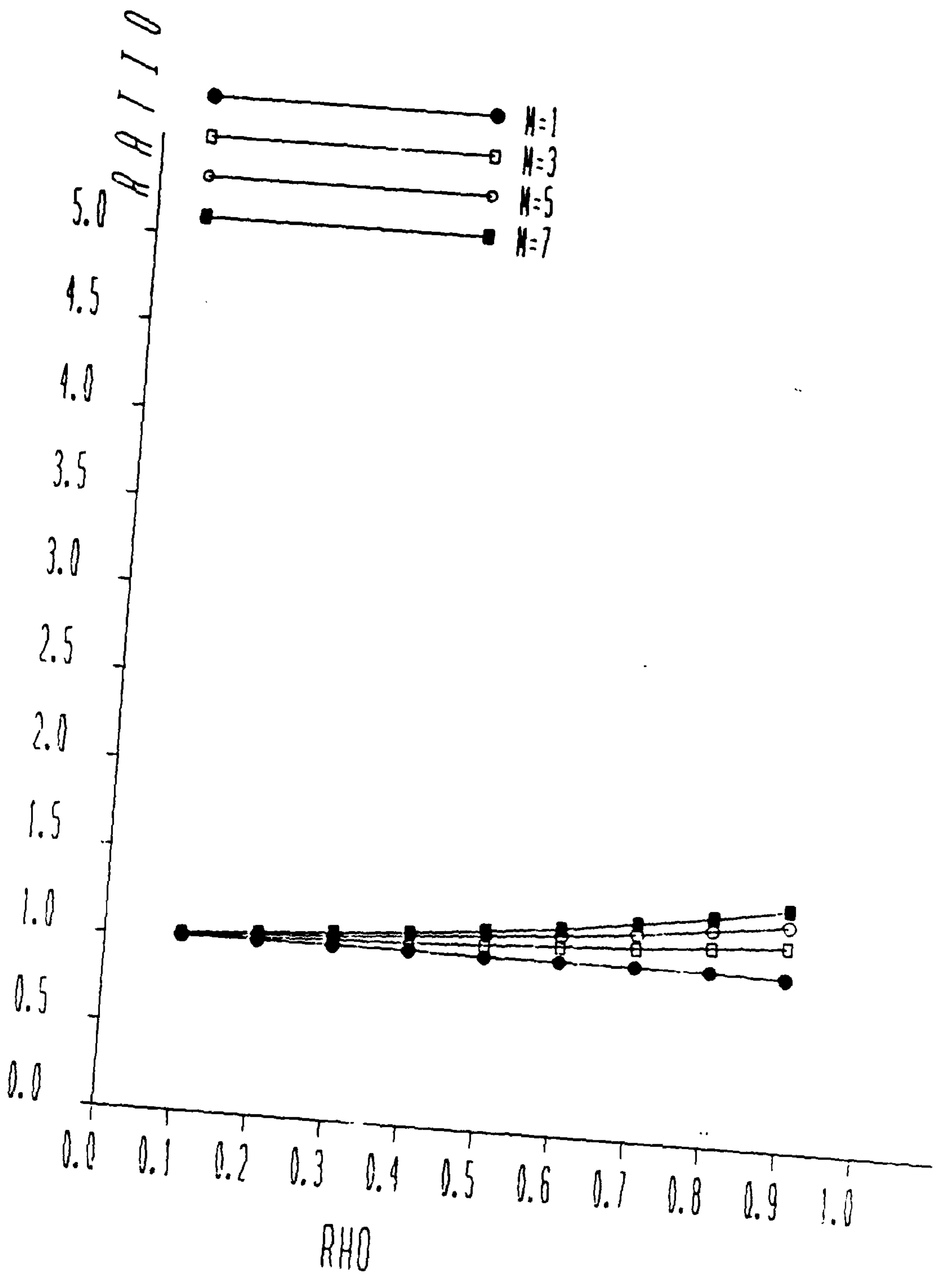


FIGURE 2 JOB/TASK

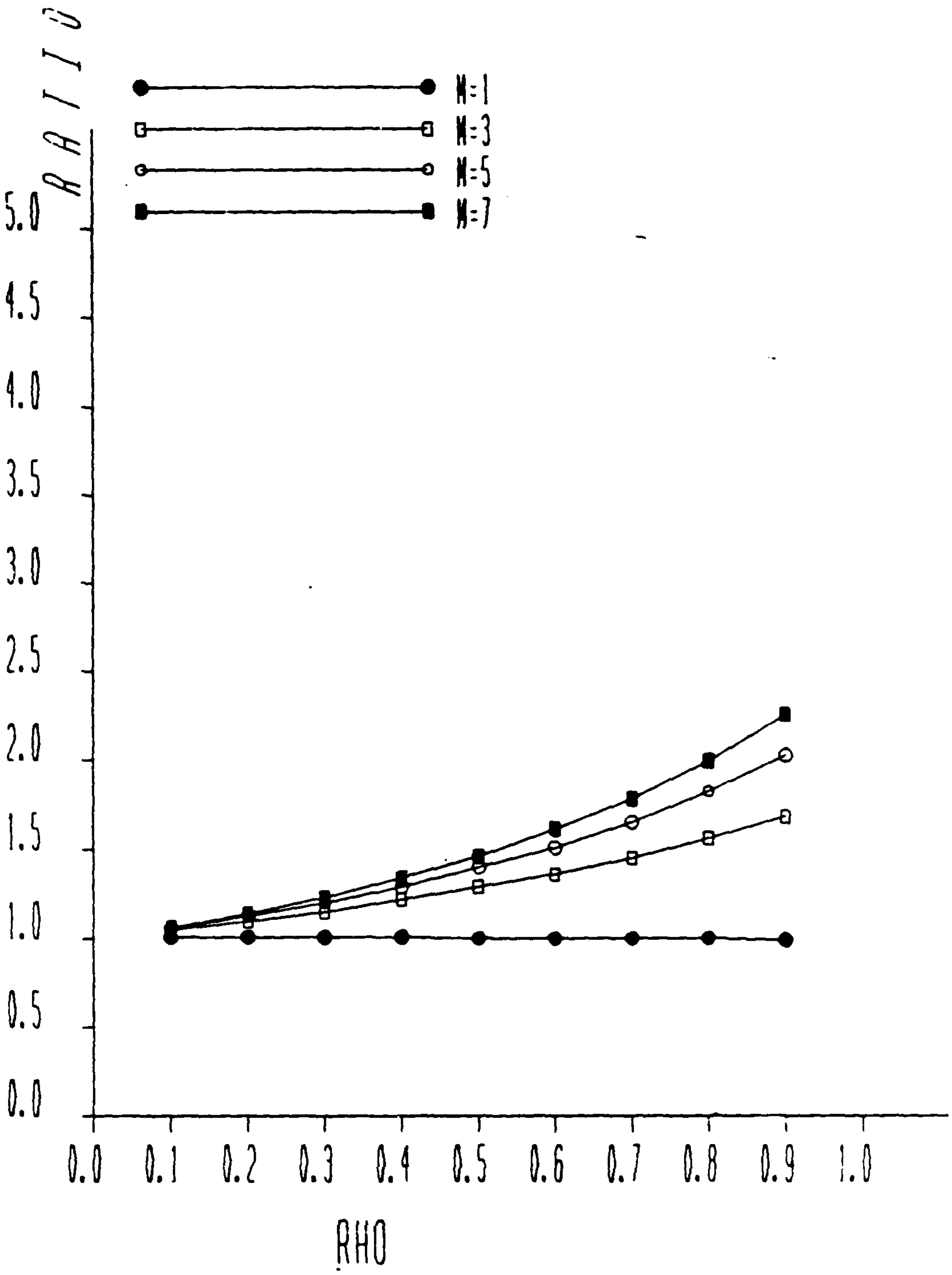
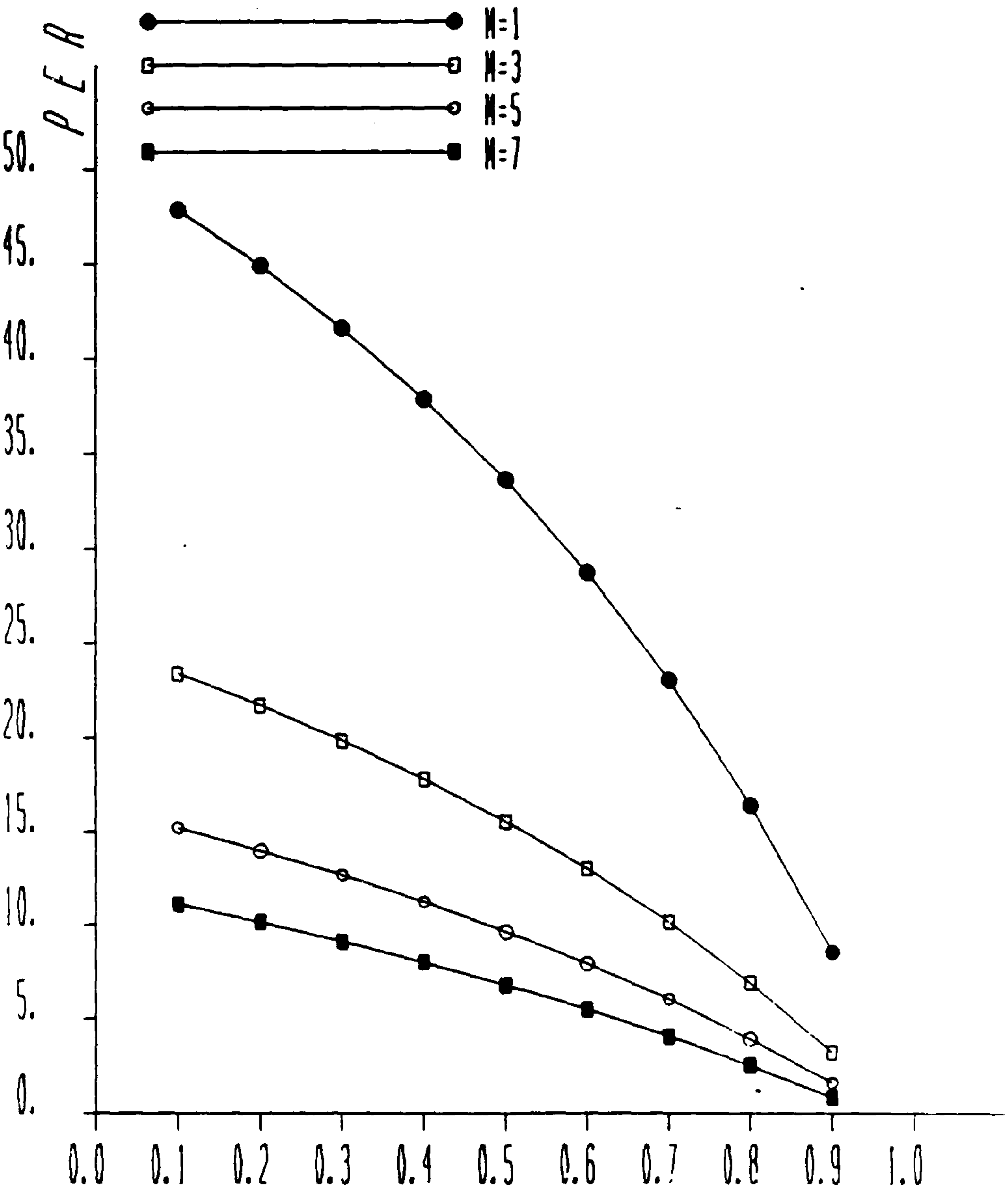


FIGURE 3 FCFS/TASK



RHO

FIGURE 4A LOWER BOUND

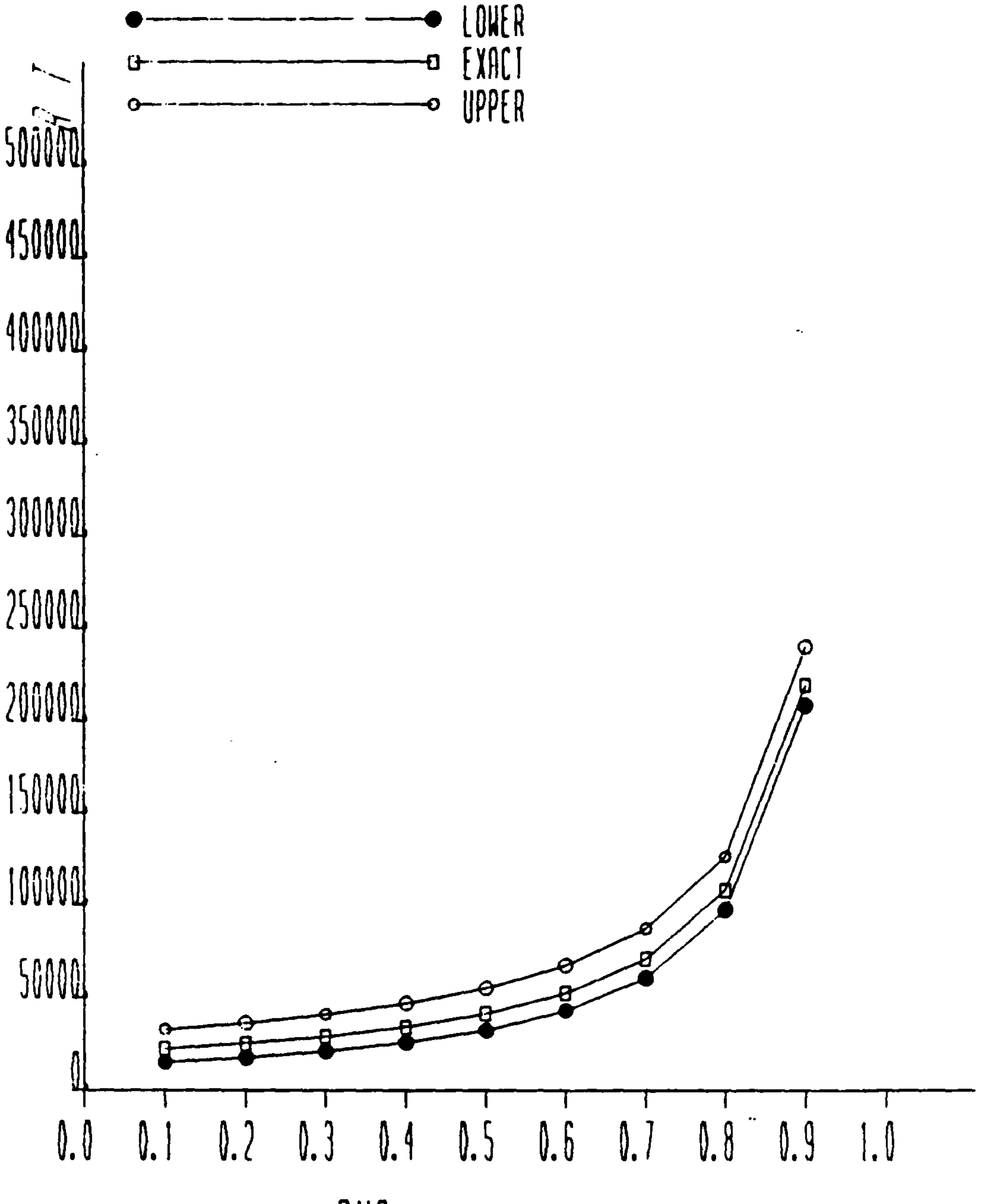


FIGURE 4B GROUP RESPONSE M=2

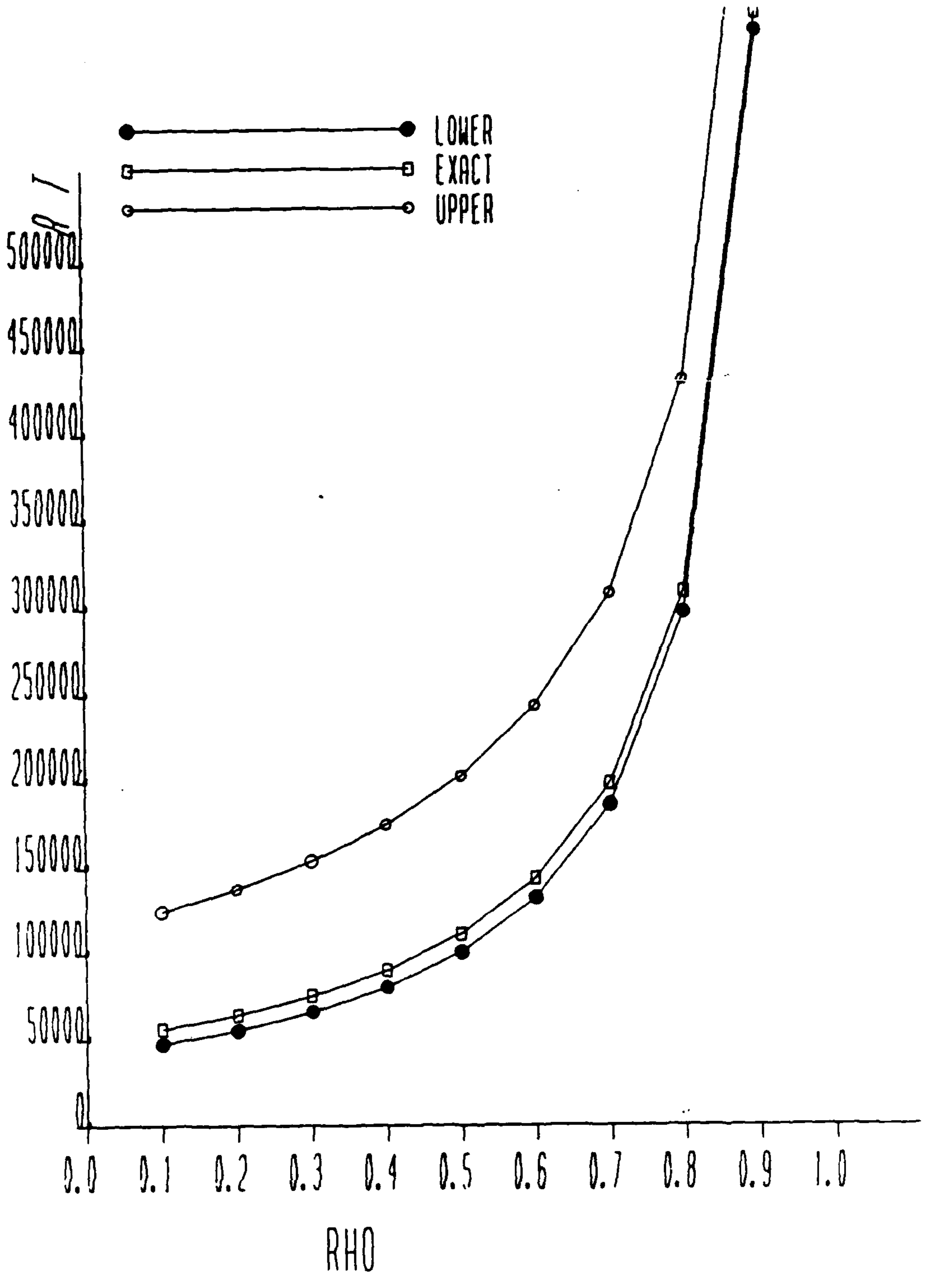


FIGURE 4C GROUP RESPONSE M=5

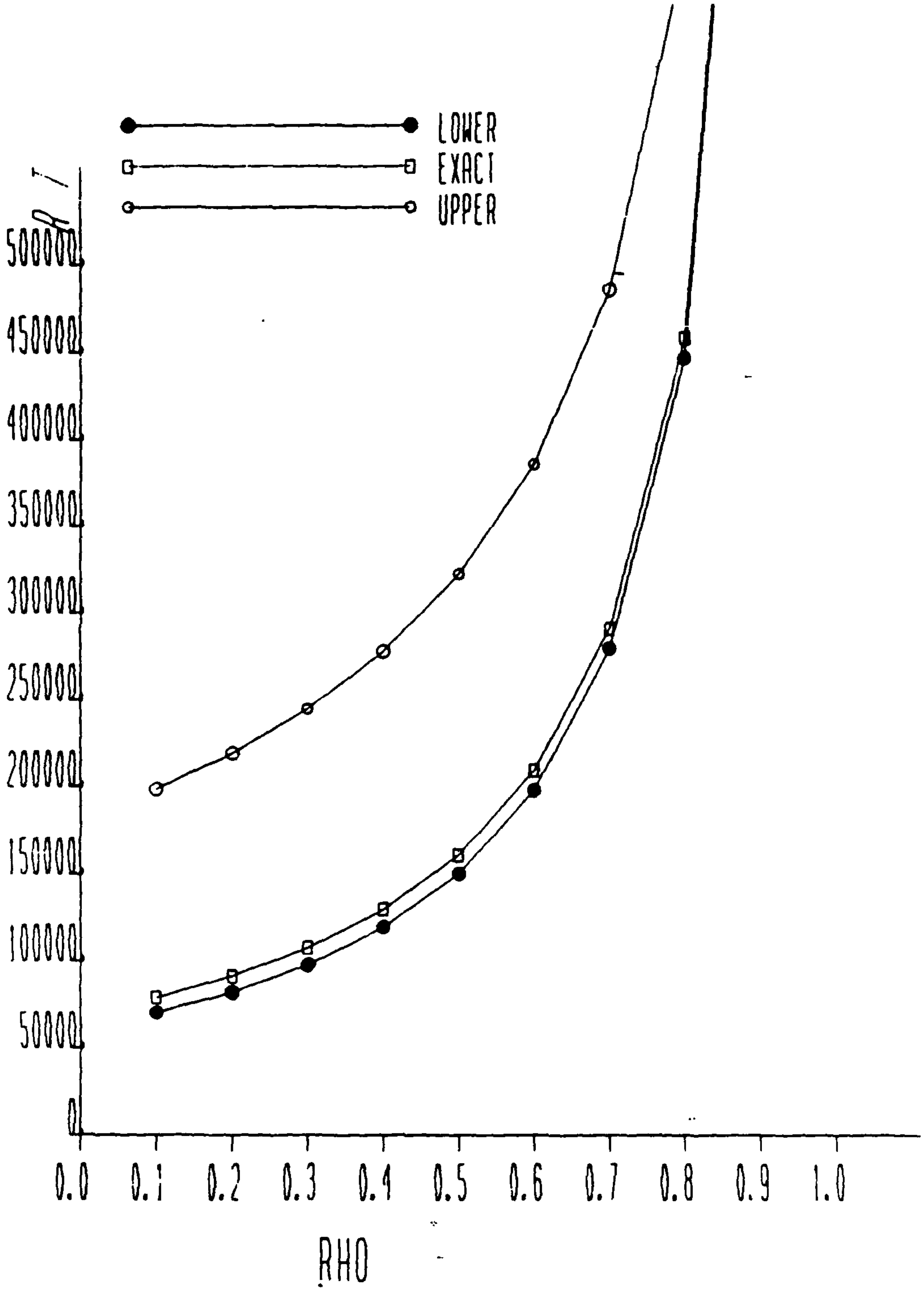


FIGURE 4D GROUP RESPONSE M=7

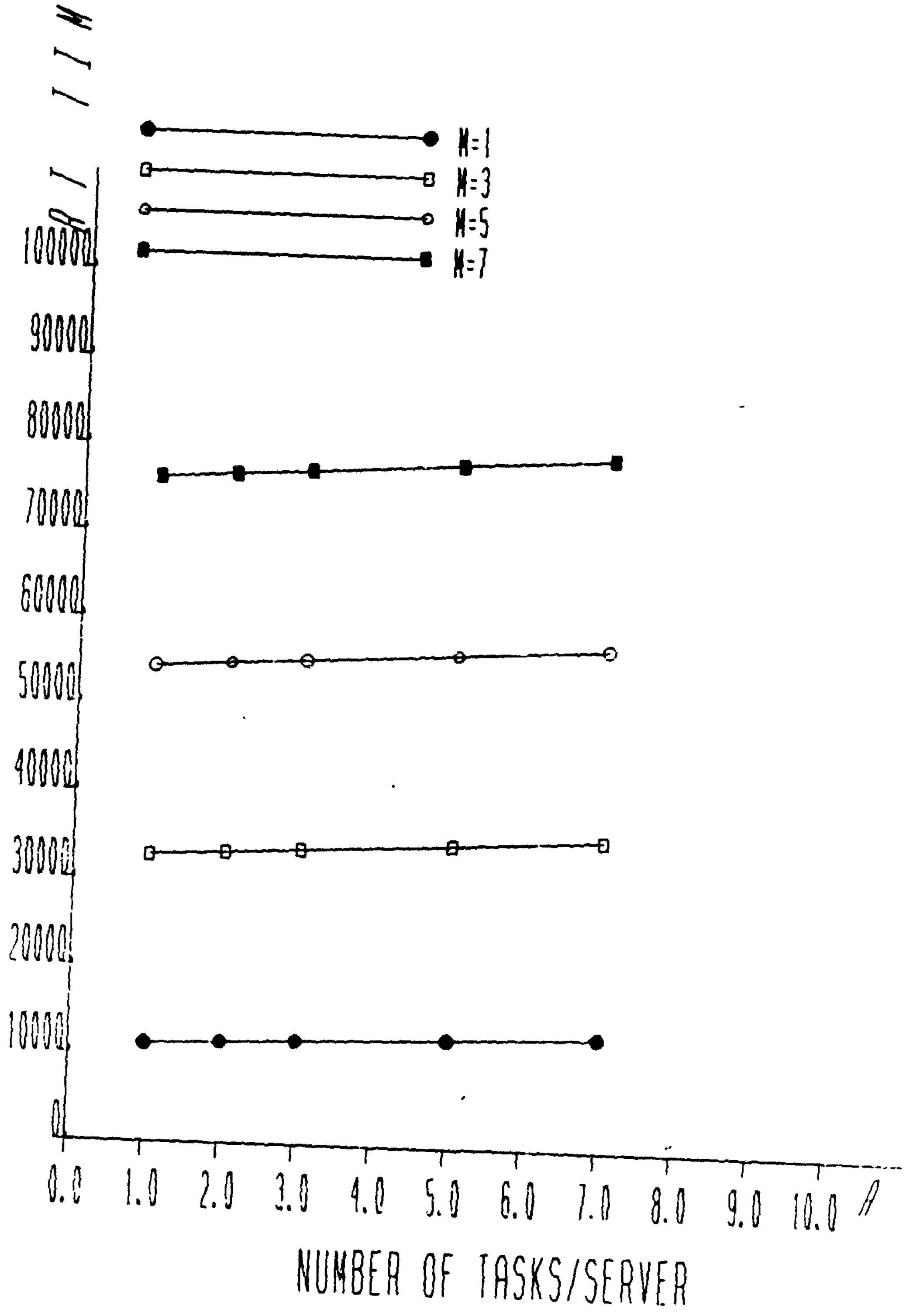


FIGURE 5A RESPONSE TIME RHO=.1

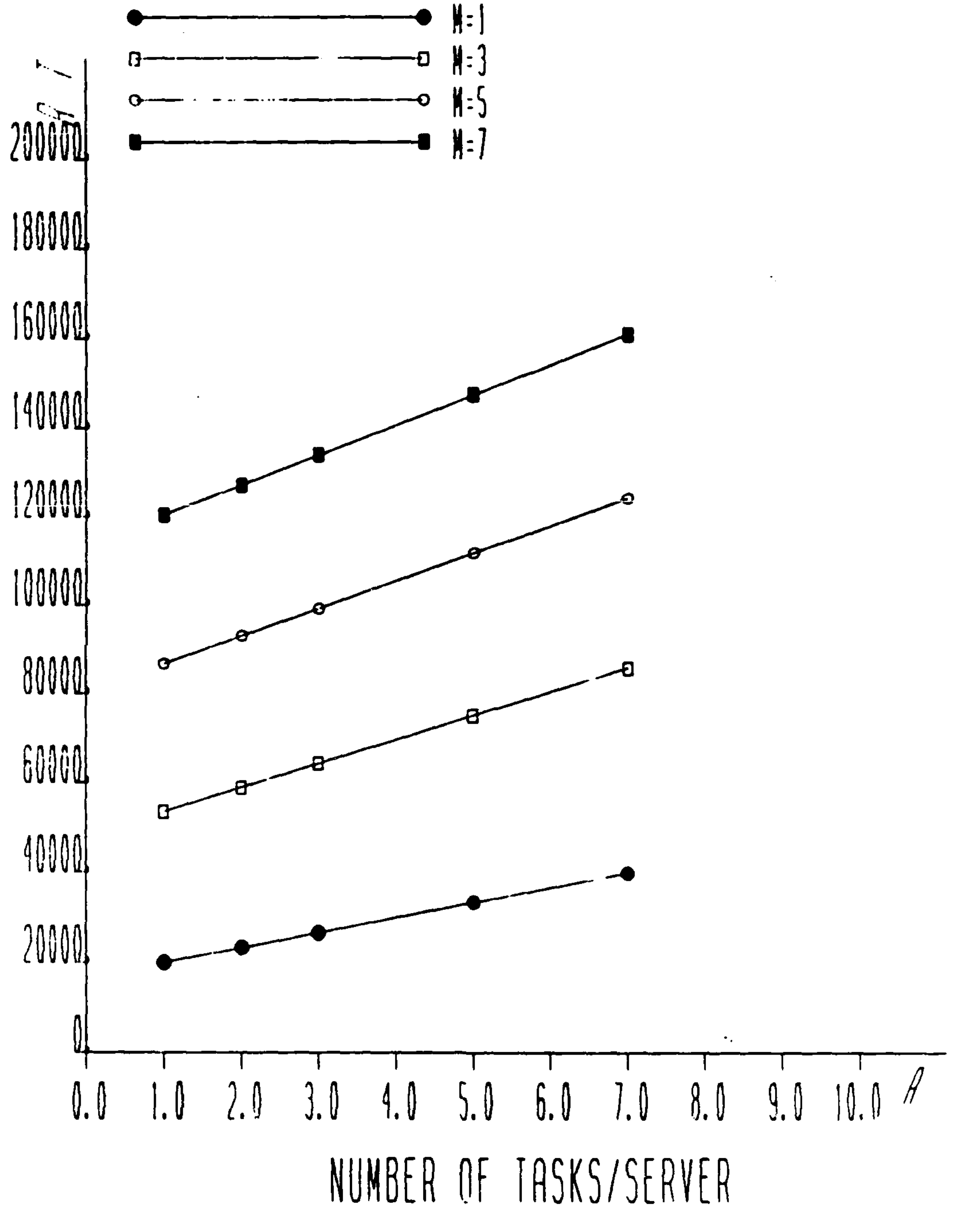


FIGURE 5B RESPONSE TIME RHO=.5

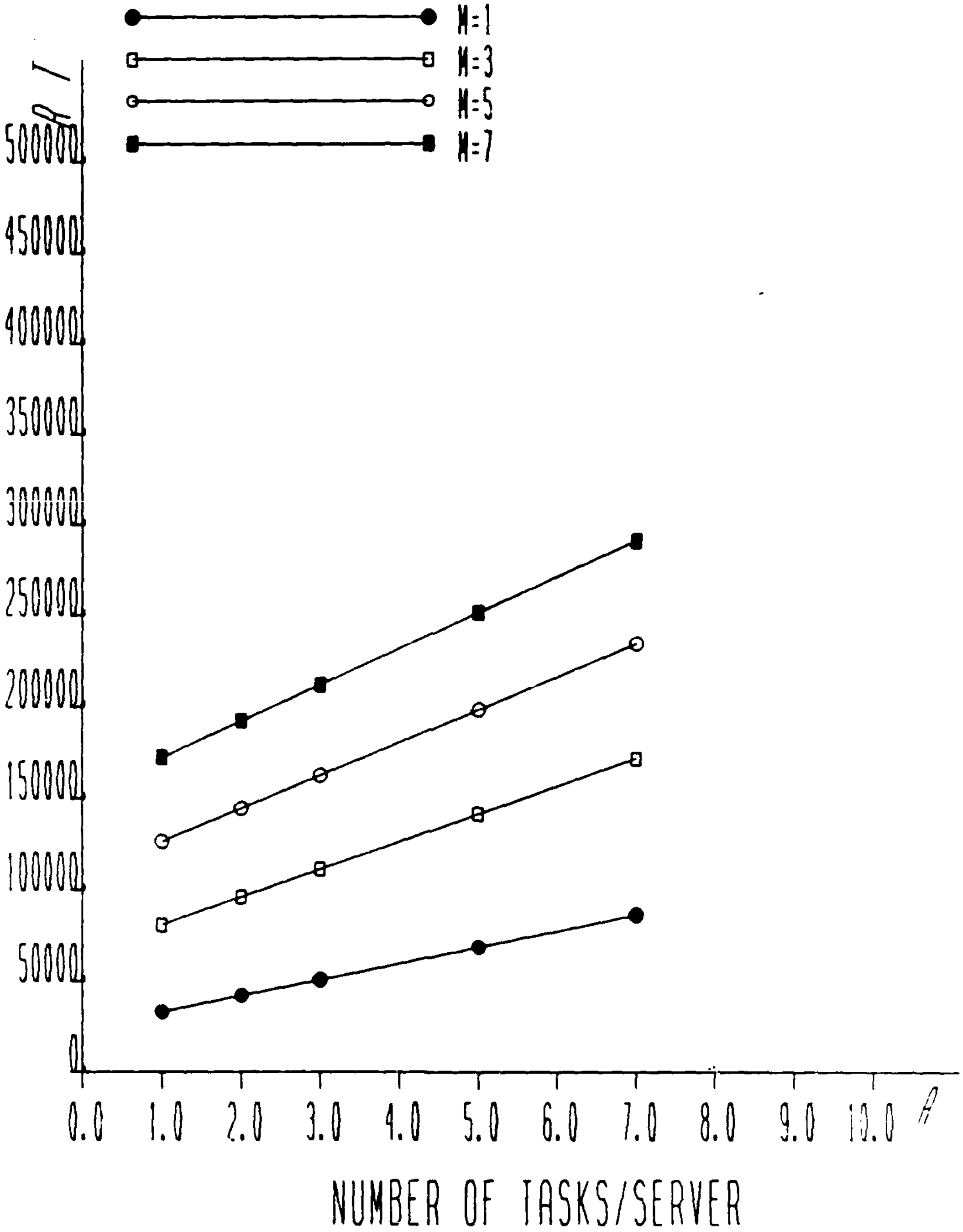


FIGURE 5C RESPONSE TIME RHO=.7

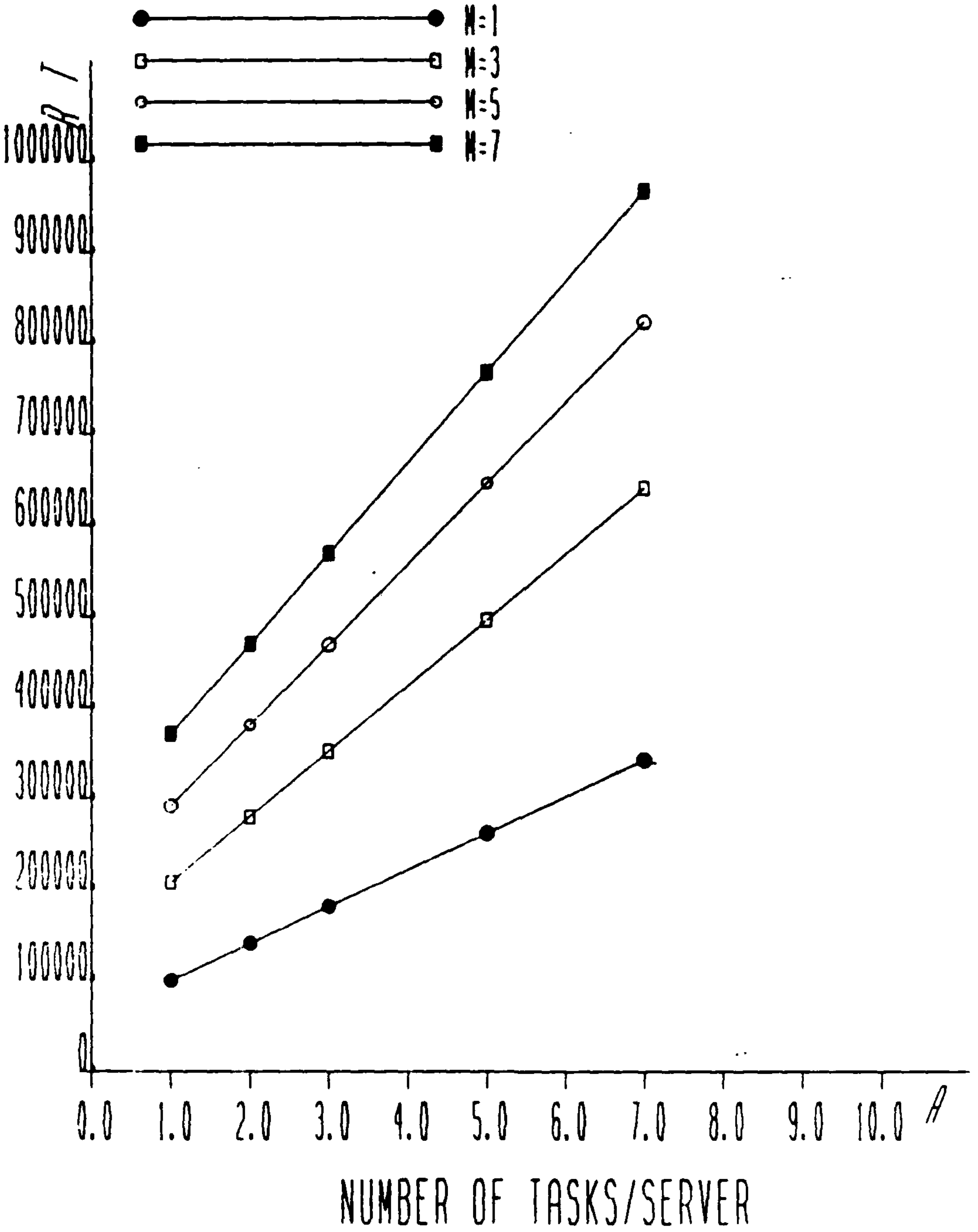


FIGURE 5D RESPONSE TIME RHO=.9

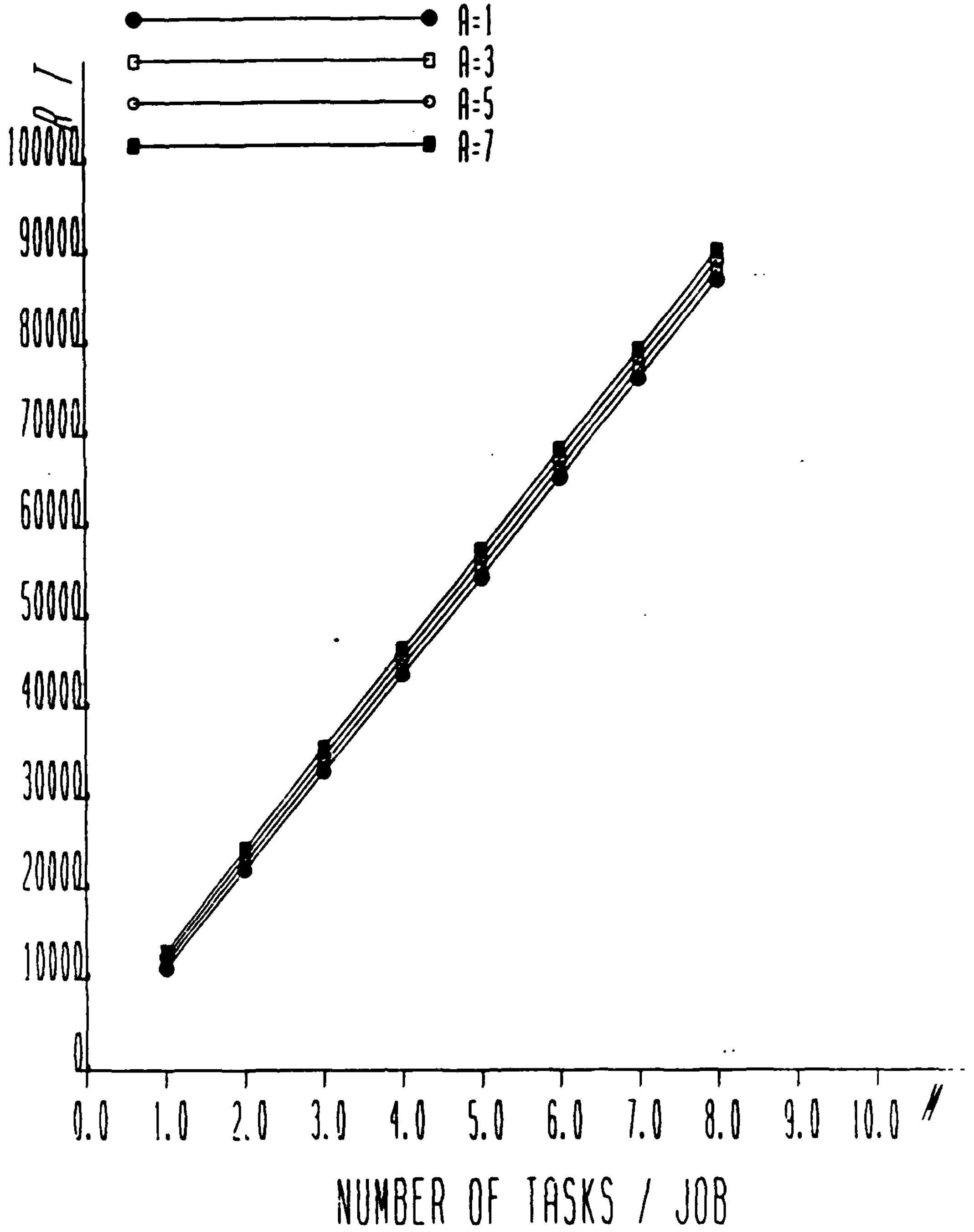


FIGURE 6A   RESPONSE TIME RHO-.1

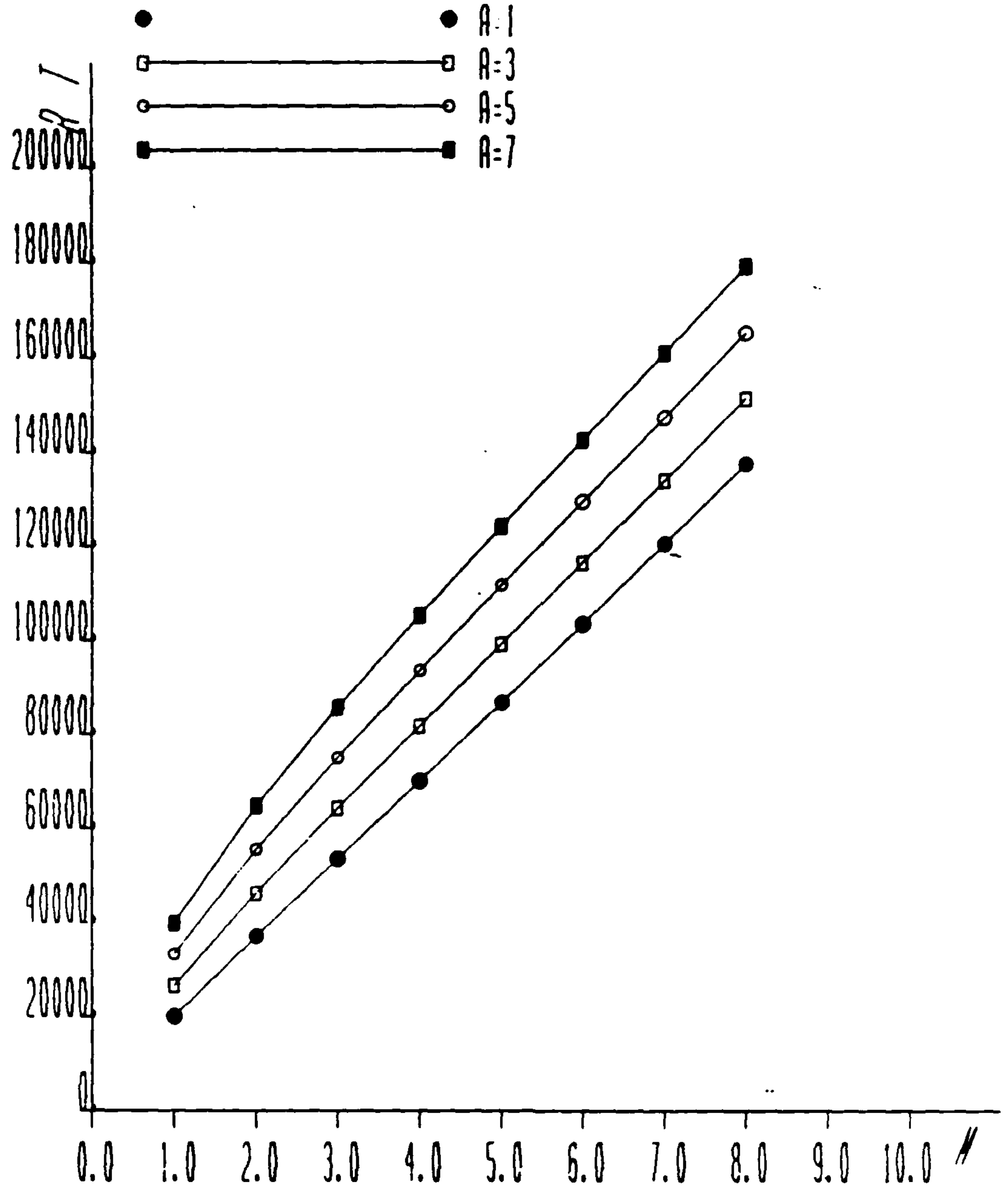
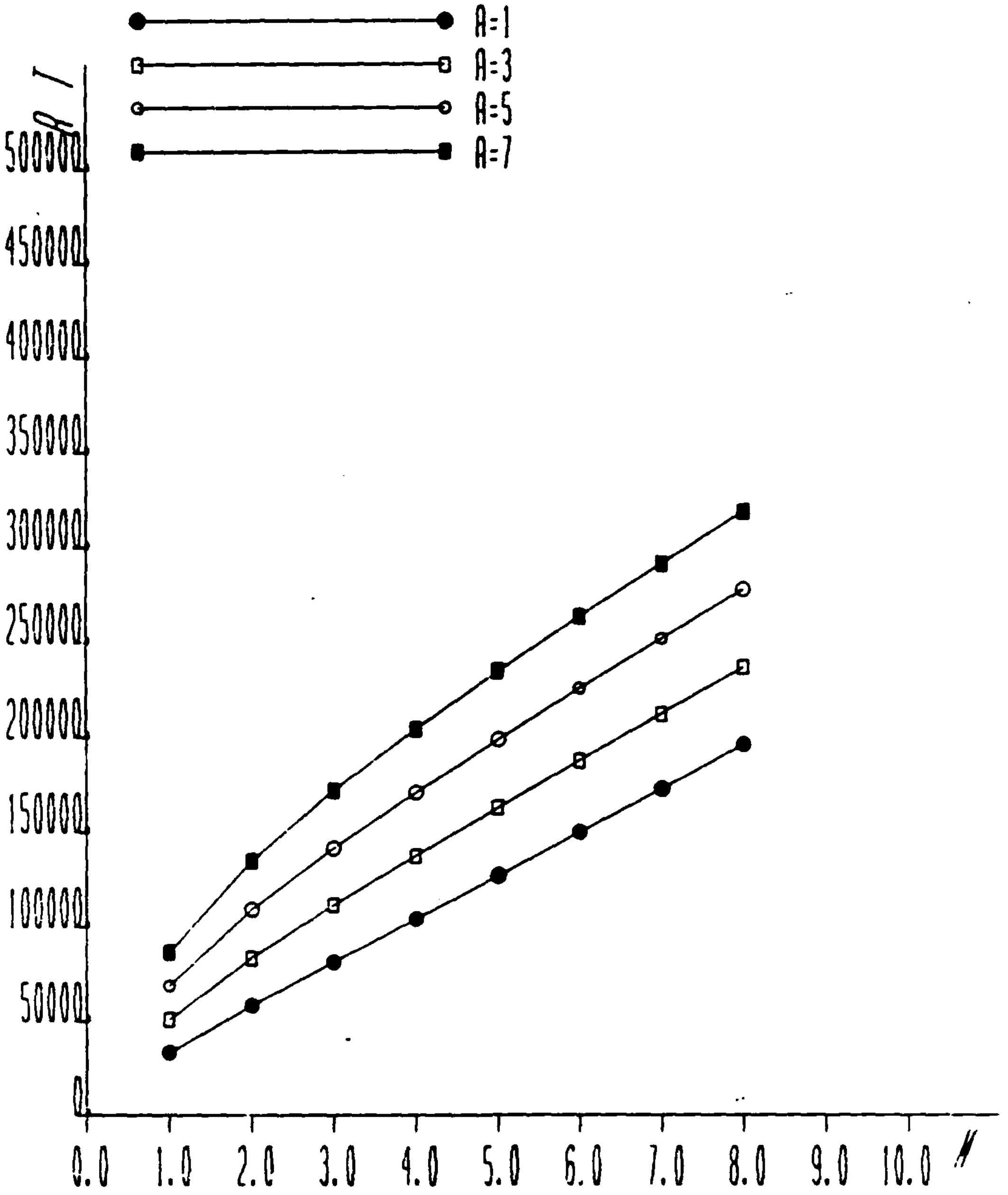


FIGURE 6B RESPONSE TIME RHO=.5



NUMBER OF TASKS/ JOB

FIGURE 6C RESPONSE TIME RHO=.7

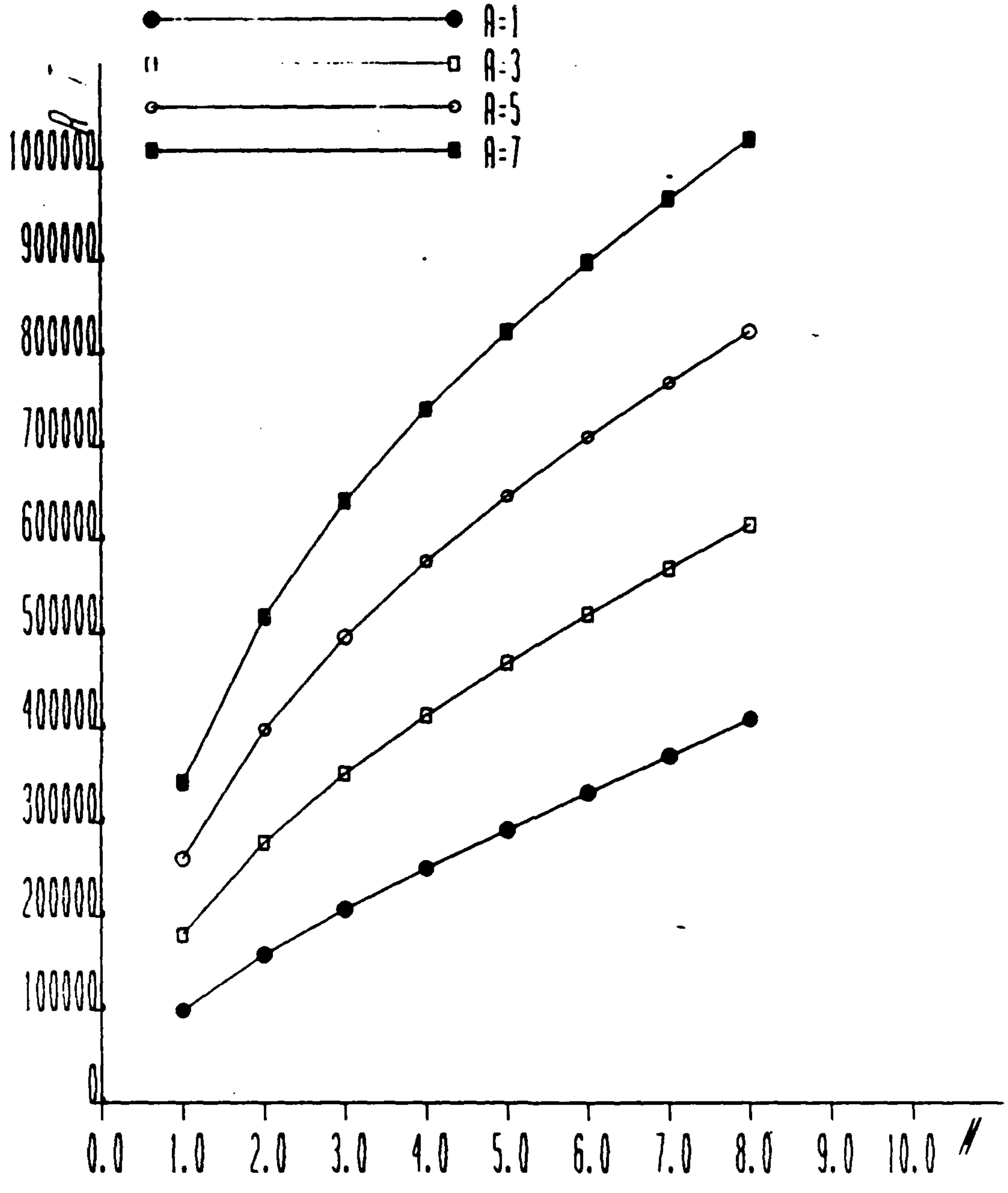


FIGURE 6D RESPONSE TIME RHO=.9

Analysis of the Effects of Delays on Load Sharing *

Ravi Mirchandaney
ECE Dept., U. Massachusetts
Amherst MA. 01003

Don Towsley
COINS Dept., U. Massachusetts
Amherst MA. 01003

John A. Stankovic
COINS Dept., U. Massachusetts
Amherst MA. 01003

February 1987

Abstract

In this paper, we study the performance characteristics of simple load sharing algorithms for distributed systems. In the system under consideration, it is assumed that non-negligible delays are encountered in transferring tasks from one node to another and in gathering remote state information. Because of these delays, the state information gathered by the load sharing algorithms is out of date by the time the load sharing decisions are taken. This paper analyzes the effects of these delays on the performance of three algorithms that we call Forward, Reverse and Symmetric. We formulate queueing theoretic models for each of the algorithms operating in a homogeneous system under the assumption that the task arrival process at each node is Poisson and the service times and task transfer times are exponentially distributed. Each of the models is solved using the Matrix-Geometric solution technique and the important performance metrics are derived and studied.

*This work was supported, in part, by the National Science Foundation under grant ECS-8406402 and by RADC under contract RI-44896X

1 Introduction

Distributed computer systems possess many potentially attractive features. Some of these are the capability to share processing of tasks in the event of overloads and failures, reliability through replication, and modularity. This study focuses on the issue of sharing computation power between nodes of a distributed system in response to imbalances in loads.

It will be assumed that tasks arrive at the nodes in a random fashion. Thus, situations can develop whereby some of the nodes are excessively busy while others are idle at the same time[LIVN82]. This kind of situation is detrimental to performance because tasks at the busy nodes experience very high waiting times, while the less busy nodes have idle cyles at the same time. The function of load-sharing is to minimize the occurrence of such scenarios by moving tasks from overloaded nodes to less busy ones.

Distributed load balancing has been an active area of research for some time. For example, Stone[STON78b,STON78a], Bokhari[BOKH79] and Towsley[TOWS86] examined static algorithms that utilized information only about the average behavior of tasks in deciding their assignments. Tantawi and Towsley[TANT85] investigated an optimal probabilistic assignment scheme. Silva and Gerla[dSeS84] determine an optimal load sharing strategy under the assumption that the nodes and the communication network can be modelled as product form queueing networks. Recently, Lee[LEE87] studied the effects of task transfer delays on simple algorithms that do not utilize any remote state information.

Eager et.al.[EAGE86] evaluated three simple load sharing schemes. They assume that the entire overhead due to load sharing is transferred onto the CPU and is modelled as an increased load on the same. Further, the nodes are assumed to be part of a local area network connected by a high bandwidth medium. Thus, there are no delays in transferring tasks and remote state information is always perfectly accurate.

While these works provided significant insight into various aspects of load sharing, the problem of communication delays and out of date state information and its impact on load sharing has not been investigated in any great detail. In this paper, we focus on the effect of communication delays upon the performance of simple load sharing algorithms. We feel that this problem is interesting in that there exist a sufficient number of system architectures that will generate significant delays in task transfers. For instance, Theimer et.al.[THEI85], report their concerns with task transfer delays. Furthermore, they have acknowledged that if the files used by a task were to be transferred (as they might have to be if the nodes were disk-based), the effect of delays would become even more prominent (the V-System is currently comprised of diskless workstations). Also, the question of how to deal with out of date state information has been one of the many interesting developments in designing algorithms for distributed systems as investigated in Stankovic[STAN85].

In this connection, we have developed analytical models that help us better understand the above issues. Various relevant performance metrics are derived from these models and the load sharing algorithms are compared on the basis of these metrics. By studying the results obtained from the model solution, we are able to determine the exact effects of delays and out of date state information on load sharing in general. Furthermore, we are able to determine the range of delays and traffic intensities over which state information is worth gathering and useful load sharing can be performed.

The remainder of this paper is organized as follows: In Section 2, we provide a brief description of the system architecture and the load sharing algorithms. Section 3 comprises the description of the Markov process corresponding to the Symmetric algorithm and its Matrix Geometric solution. The analysis corresponding to the Forward and Reverse probing algorithms will only be described in brief. This is because the analysis of Symmetric subsumes that of Forward and Reverse. In Section 4, we describe the important results of this research and we summarize our work in Section 5. Finally, Appendices A and B describe the internals of the matrices involved with the solution of the Markov processes.

2 System Architecture and Load Sharing Algorithms

2.1 System Architecture and Motivation

Processing and transmission of communication messages for state updates (probes) and for tasks can potentially generate considerable overhead at the nodes. Different system architectures can impose very different costs for these overheads. At one end of the spectrum, nodes can have dedicated processors to handle communication overheads, supported by a very high bandwidth fiber-optic bus communication. On the other end of the spectrum, nodes can be multiplexed between application tasks and communication packet processing.

We have made the following assumptions about the system that we will be considering. The architecture of the individual nodes includes a powerful Bus Interface Unit(BIU), which is used to process most of the overhead generated by task and probe movement. For instance, the BIU will have a DMA capability to access main memory without much interference to the CPU.

While the bulk of the overhead processing for task transfer is transferred to the BIU, delays will nevertheless occur during this processing. There will also be network delays in the transmission of probes and tasks. We are interested in studying the combined effects of these delays. Furthermore, we believe that it is reasonable to assume that the relative sizes of tasks and probes will be quite different. The physical transfer of a task may require tens of communication packets, while a probe or a response to one would in all likelihood need at most one packet. Thus, it is reasonable to imagine a ratio of 50:1 or more in the relative sizes of tasks vs. probes. Consequently, it appears that the delays incurred by tasks in the BIU's and the network will be significantly larger than those incurred by the probes. In our analysis, the delays incurred by probes will be assumed to be negligible when compared with those incurred by task transfers.

2.2 Load Sharing Algorithms

The three algorithms that we have studied in the context of this research are called Forward, Reverse and Symmetric. Each algorithm is provided with a threshold T. The algorithms are described in the following few paragraphs.

- **Forward:** The algorithm is activated each time a local task arrives at the node. If the number of tasks at this node (including the task currently being executed) is greater than $T + 1$, an attempt is made to transfer the newly arrived task to another node. A finite number, L_p, of nodes (usually $L_p = 2$ or 3 is adequate) is probed at random to determine a placement for the task. A probed node responds positively if the number of tasks it possesses is less than $T + 1$ and it is not already waiting for some other remote task. If more than one node responds positively, the sender node transfers the task to one of these respondents, picked at random. If none of the probed nodes responds positively, i.e., this probe was unsuccessful, the node waits for another local arrival before it can probe again.

- **Reverse:** This algorithm is activated every time a task completes at a node and the total number of tasks at the node is less than $T + 1$ and the node is not already waiting for a remote task to arrive. If so, the node probes a subset of size L_p remote nodes at random to try and acquire a remote task. Only nodes that possess more than $T + 1$ tasks, (including the currently executing one) can respond positively. If more than one node can transfer a task, the probing node chooses one of these at random from which it requests a task.

- **Symmetric:** This algorithm combines the two schemes of Forward and Reverse. Thus, if a node goes above $T + 1$ upon the arrival of a local task, it attempts to transfer a task and if it drops below $T+1$ upon a task completion, it attempts to acquire a remote task.

In all the algorithms described above, it is assumed that probing takes zero time. This is based on the initial assumption that probes are much smaller entities than are tasks. Thus, the overhead for processing a probe at the BIU is much smaller than for tasks. Further, probes occupy much less of the communication bandwidth than tasks. Thus, the entire delay is assumed to occur during actual task transfer. Furthermore, we have seen in separate studies (not described in this paper) that as long as the ratio of task transfer times to probe transfer times is sufficiently large (≥ 50), the system essentially behaves as if the probes actually take zero time. We are currently investigating this phenomenon in greater detail.

3 Mathematical Analysis

It is assumed that the task arrival process at each node is Poisson, with parameter λ. Also, the service times and task transfer times are assumed to be exponentially distributed, with means $1/\mu$ and $1/\gamma$, respectively. The task transfer time includes the time between the initiation of a transfer from a node and the successful reception of the task at the destination node. The nodes are assumed to be homogeneous, i.e., the nodes have identical processing power and the arrival process at each node is the same. Tasks are assumed to be executed on a First-Come-First-Served (FCFS) basis at each node.

Let $N_t^{(i)}$ be the number of tasks at node i at time t and $J_t^{(i)}$ be the probe state of node i, at time t. The probe state indicates whether the node is probing or being probed, etc. For example, in a system of M nodes, the instantaneous state of the network can be represented by the 2M-tuple

$$(N_t^{(1)}, N_t^{(2)}, ..., N_t^{(M)}; J_t^{(1)}, J_t^{(2)}, ...J_t^{(M)})$$

If the probe state $J_t^{(i)}$ is defined appropriately then, due to the Poisson arrival assumption and the exponential service and task transfer times, the process corresponding to the above state description is Markovian.

It is clear that the model has a very large state space and is difficult to solve, even for moderately sized systems. Consequently, we decompose the model such that the model for each node can be solved independently of the others [EAGE86]. The interactions between the nodes which result in task transfers for the purpose of load sharing in the distributed system, are modelled by means of modifications to the arrival and/or departure process at each node. These interactions will be described in detail later in this subsection.

We conjecture that the method of decomposition is asymptotically exact as the number of nodes tends to infinity. Actual experimental results indicate that there exists very good agreement between the model and simulations even when the systems are of relatively small size (= 10 nodes). Thus, the approximation is likely to be even better for larger systems.

The analysis of the algorithms is performed using the Matrix-geometric solution technique [NEUT81] which yields an exact solution of the model for each node. The model for the Symmetric probing algorithm will be described in detail. However, the analysis of the Forward and Reverse algorithms will only be described in brief, with a presentation of the main performance metrics.

The material in this paper involves several Jacobi matrices, whose detailed definitions will be provided as in Latouche[LATO81]. A matrix such as

$$\begin{bmatrix} b_0 & c_0 & 0 & 0 & & & \cdots & \\ a_1 & b_1 & c_1 & 0 & & & \cdots & \\ 0 & a_2 & b_2 & c_2 & & & & \\ & & & & & & & \\ \cdots & \cdots & & & a_{m-2} & b_{m-2} & c_{m-2} & 0 \\ \cdots & \cdots & & & 0 & a_{m-1} & b_{m-1} & c_{m-1} \\ \cdots & \cdots & & & 0 & 0 & a_m & b_m \end{bmatrix}$$

will be displayed as

$$\left\| \begin{array}{ccccccc} & c_0 & c_1 & \cdots & c_{m-3} & c_{m-2} & c_{m-1} \\ b_0 & b_1 & b_2 & \cdots & b_{m-2} & b_{m-1} & b_m \\ a_1 & a_2 & a_3 & \cdots & a_{m-1} & a_m & \end{array} \right\|$$

Figure 1 represents the state diagram for the Symmetric algorithm operating at a single node using an arbitrary threshold T. The state of the node is represented by a tuple (N_t, J_t), where N_t is the number of tasks at a node and J_t is the probe state that indicates if the node is either probing, being probed, neither of the above, or both. The probe states have the following codes:

- 0 : if not probing and not being probed,
- 1 : if reverse probing,
- 2 : if being forward probed,
- 3 : if reverse probing and being forward probed.

The actual representation of this process takes the form of an infinite cylinder. However, for ease of description, we have chosen to open out this cylinder and consequently, the row corresponding to probe state 3 is duplicated, once as the top row and again as the bottom row in Figure 1. In 3-Space, the top and the bottom rows would be merged together.

We define

$$y(n,j) = \lim_{t\to\infty} P(N_t = n,\ J_t = j),\ 0 \le n,\ 0 \le j \le 3,$$

$$\mathbf{p}_n = (y(n,0), y(n,1), y(n,2), y(n,3)),\ 0 \le n,$$

$$\vec{\mathbf{p}} = (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \ldots\ldots \mathbf{p}_i, \ldots\ldots).$$

If the Markov process (N_t, J_t) is ergodic then $\vec{\mathbf{p}}$ is its steady state probability vector satisfying $\vec{\mathbf{p}}Q = 0$, where Q is the infinitesimal generator of this Markov process. Q_S, the infinitesimal generator for the Symmetric algorithm, has the structure of a block-tridiagonal matrix of the form

$$Q_S = \left\| \begin{array}{ccccccccc} & B_{01} & \ldots & B_{01} & B_{01} & A_0 & A_0 & \ldots \\ B_{00} & B_{11} & \ldots & B_{11} & B_{21} & A_1 & A_1 & \ldots \\ B_{10} & B_{10} & \ldots & B_{10} & A_2 & A_2 & A_2 & \ldots \end{array} \right\|$$

where we define the matrices $B_{00}, B_{01}, B_{10}, B_{11}, B_{21}, A_2, A_1$ and A_0 in Appendix A.

In the subsequent discusssion, h is the probability of failure in finding an assignment for a spare task in response to a set of forward probes. Thus, $\bar{h} = 1 - h$, is the probability that at least one of the probed nodes will accept a remote task. Also, q is the probability of failure in finding a remote task for a set of reverse probes, and $\bar{q} = 1 - q$.

The effect of a node sending a forward probe when it goes above $T+1$ is represented by the transition λh. When the node makes a transition anywhere below $T+1$ on the completion of a task, it sends out reverse probes in order to get a remote task. A successful transition is represented by $\mu\bar{q}$ and an unsuccessful set of reverse probes is represented by the transition μq.

Thus, on the completion of a task when the node goes below $T+1$, it sends out reverse probes, if it is not already waiting for a remote task to arrive in response to an earlier reverse probe. A transition of this type is represented from $(n,0)$ to $(n-1,1)$ or $(n,2)$ to $(n-1,3)$, where $0 < n \le T+1$. When a remote node sends a forward probe into this node, it makes the transition from $(n,0)$ to $(n,2)$ or $(n,1)$ to $(n,3)$, where $0 \le n \le T$. This means that the remote node is going to transfer a task to this node, on the basis of a successful probe. The rate of receiving forward probes is denoted by α. The rate at which this node sends out tasks in response to nodes that asked it for tasks is μ'. Thus, the rate at which a node makes the transition (n, j) to $(n-1, j)$, for $n \geq T+2$ equals $\mu + \mu'$.

As can be seen from the generator Q_S, the Markov process has a regular structure comprised of the A_0, A_1 and A_2 matrices, preceded however by the irregular boundary conditions. The size of the irregular portion of the matrix depends upon the threshold at which the process is operating. There will be exactly $T-1$ columns of the matrices (B_{01}, B_{11}, B_{10}).

Neuts[NEUT81] examined Markov processes with such generators and determined the conditions for positive recurrence when the infinitesimal generator $A = A_0 + A_1 + A_2$, corresponding to the geometric part of the Markov Process, is irreducible. However, for our problem, A is lower triangular and reducible. In such cases, the stability criterion has to be determined explicitly.

Consider the non-linear matrix equation

$$A_0 + RA_1 + R^2A_2 = 0$$

such that R is its minimal non-negative solution. It can be shown that R is lower triangular, given the structure of A_0, A_1 and A_2 [NEUT81]. Furthermore, $R = [r_{i,j}]$, where

$$\begin{aligned}
r_{i,j} &= 0, \forall i < j \\
r_{1,1} &= \frac{\delta - (\delta^2 - 4(\mu + \mu')\lambda h)^{1/2}}{2(\mu + \mu')} \\
r_{2,2} &= \frac{\delta + \gamma - ((\delta + \gamma)^2 - 4(\mu + \mu')\lambda h)^{1/2}}{2(\mu + \mu')} \\
r_{3,3} &= r_{2,2} \\
r_{4,4} &= \frac{\delta + 2\gamma - ((\delta + 2\gamma)^2 - 4(\mu + \mu')\lambda h)^{1/2}}{2(\mu + \mu')} \\
r_{2,1} &= \frac{\gamma}{\delta - (r_{1,1} + r_{2,2})(\mu + \mu')} \\
r_{3,1} &= r_{2,1} \\
r_{3,2} &= 0 \\
r_{4,1} &= \frac{(r_{4,2}r_{2,1} + r_{4,3}r_{3,1})(\mu + \mu')}{\delta - (r_{1,1} + r_{4,4})(\mu + \mu')} \\
r_{4,2} &= \frac{\gamma}{\delta + \gamma - (r_{2,2} + r_{4,4})(\mu + \mu')} \\
r_{4,3} &= r_{4,2}
\end{aligned}$$

where $\delta = (\lambda h + \mu + \mu')$.

Thus, the diagonal elements of R can be written explicitly in terms of the parameters of the Markov process. Once the diagonal elements are determined, the elements below the diagonal are computed recursively from the solution of the diagonal elements.

By adapting Theorem 1.5.1 from Neuts[NEUT81], the Markov process Q_S is positive recurrent if and only if $sp(R) < 1$ and the matrix M (defined below) possesses a positive left invariant probability vector. Because R is lower triangular, its eigenvalues are its diagonal elements. One can show that $sp(R) < 1$ if

$$\lambda h < \mu + \mu'$$

The matrix M, given by

$$\left\| \begin{array}{ccccc} & B_{01} & \dots & B_{01} & B_{01} \\ B_{00} & B_{11} & \dots & B_{11} & B_{21} + RA_2 \\ B_{10} & B_{10} & \dots & B_{10} & \end{array} \right\|$$

is an irreducible, aperiodic matrix. The second condition holds because of the irreducibility of M. The vector $(\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{T+1})$ is the left eigenvector of M.

Intuitively, the stability condition means that the rate of processing tasks (including the ones that are sent out of this node) is greater than the total arrival rate of tasks into this node. Thus, on the average, whenever there are more than $T + 1$ tasks at a node, the process drifts towards the boundary specified by the threshold T. Similar analysis may be carried out for the Forward and Reverse probing algorithms, with the appropriate substitution of parameters.

We now assume that all the values of all the parameters are known. First, the boundary conditions are determined, by solving a system of linear equations. Thus, for an arbitrary threshold T, we have

$$(\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{T+1}) \left\| \begin{array}{ccccc} & B_{01} & \dots & B_{01} & B_{01} \\ B_{00} & B_{11} & \dots & B_{11} & B_{21} + RA_2 \\ B_{10} & B_{10} & \dots & B_{10} & \end{array} \right\| = 0$$

where the number of columns in the matrix is exactly $T + 1$. We know from Neuts[NEUT81] that if the process is positive recurrent

$$\mathbf{p}_i = \mathbf{p}_{T+1} R^{T+1-i}, \forall i \geq T + 1$$

Thus,

$$\sum_{i \geq T+1} \mathbf{p}_i = \mathbf{p}_{T+1}(I-R)^{-1}$$

Also,

$$[\sum_{i=0}^{T} \mathbf{p}_i + \mathbf{p}_{T+1}(I-R)^{-1}]e = 1$$

$E[N]$, the expected number of tasks at a node, and $E[D]$, the expected response time of a task, are given by the following expressions:

$$\begin{aligned} E[N] &= \sum_{i \geq 1} i\, \mathbf{p}_i\, e \\ &= \mathbf{p}_{T+1}(I-R)^{-2}e + T * [\mathbf{p}_{T+1}(I-R)^{-1}e] + \sum_{i=1}^{T} i\mathbf{p}_i e \\ E[D] &= \frac{(E[N] + \frac{(Total-Flow-In)}{\gamma})}{\lambda} \end{aligned}$$

where Total-Flow-In is the flow into a node of remote tasks due to forward and reverse probes. In the next subsection, we derive the equations required to determine the values of the unknowns h, q, μ' and α and describe the iterative algorithm used to solve the resulting model.

3.1 Computational Procedure

Initially, it is assumed that the values for h, q, μ' and α are known and the model is solved using these values. In a typical step, a model solution is used to derive new values for h, q, μ' and α, and a new solution is computed. The iteration procedure that we use is described in a step-wise form, after the following definitions.

- $FFRO$: Flow rate out of tasks, as a result of forward probes made by this node.
- $FFRI$: Flow rate in of tasks, as a result of forward probes made to this node by other nodes.

- *RFRO* : Flow rate out of tasks, as a result of reverse probes made by other nodes to this node.
- *RFRI* : Flow rate in of tasks, as a result of reverse probes made by this node.

Let i denote the iteration count. Thus, $h^{(i)}, q^{(i)}, \mu'^{(i)}, \alpha^{(i)}, FFRO^{(i)}, RFRO^{(i)}$ denote the value of the variables after the i-th iteration.

Iteration Procedure

1. Let $i = 0$; choose values for $h^{(0)}, q^{(0)}, \mu'^{(0)}, \alpha^{(0)}, FFRO^{(0)}, RFRO^{(0)}$
2. Determine $Q^{(i)}$ from $h^{(i)}, q^{(i)}, \mu'^{(i)}, \alpha^{(i)}$
3. Determine $R^{(i)}$
4. Solve the linear system corresponding to the boundary conditions
5. Determine $FFRO^{(i+1)}$ and $RFRO^{(i+1)}$ from the model solution
6. If $ABS(FFRO^{(i+1)} - FFRO^{(i)}) \leq \epsilon$ and $ABS(RFRO^{(i+1)} - RFRO^{(i)}) \leq \epsilon$, where ϵ is an arbitrary small number, stop, else
7. Let $i = i + 1$. Go to 2

We have observed from experiments that the solution was insensitive to the initial values chosen for the unknown quantities. Consequently, we conjecture that there exists a unique solution to the model. Further, the number of iterations was usually small, ranging between 10 and 30.

Because of the assumption of homogeneity and because of the symmetric nature of the algorithm

$FFRO = FFRI$ and, $RFRO = RFRI$.

To determine α, we use the relation $FFRO = FFRI$, where

$$\begin{aligned} FFRO &= \lambda \bar{h} \sum_{i>T} \mathbf{p}_i \, e, \\ FFRI &= \alpha \sum_{i \leq T} \mathbf{p}_i \, [1100]^T, \end{aligned}$$

Here, h can be represented as $h = x^{L_p}$ where, L_p is the number of nodes that are probed and x is the probability that a particular node will respond negatively to a forward probe. This is given as

$$x = \sum_{i \leq T} \mathbf{p}_i \, [0011]^T + \sum_{i > T} \mathbf{p}_i e.$$

Also, $x = 1 - x$ and $\bar{h} = 1 - h$.

Thus,

$$\alpha = \frac{FFRO}{\sum_{i \leq T} \mathbf{p}_i \, [1100]^T}$$

To determine μ', we use the relation $RFRI = RFRO$, as follows:

$$RFRI = \sum_{i \geq 0} \mathbf{p}_i \, [0101]^T \gamma,$$

where $1/\gamma$ is the mean delay in receiving a remote task. Thus, $RFRI$ denotes the total flow in due to reverse probes made by this node.

$$RFRO = \mu' \sum_{i > T+1} \mathbf{p}_i \, e$$

Thus,

$$\mu' = \frac{RFRI}{\sum_{i > T+1} \mathbf{p}_i \, e}$$

To determine q, the probability of a set of reverse probes resulting in failure, we use the following procedure:

Let

$$y = \sum_{i \leq T+1} \mathbf{p}_i \, e$$

If the node probes L_p nodes to receive a remote task, then the probability that all of them will be unsuccessful is denoted by: $q = y^{L_p}$, and $\bar{q} = 1 - q$ is the probability that at least one of the reverse probes is successful.

3.2 Forward and Reverse

As mentioned in section 1, we will only briefly describe the analysis for the Forward and Reverse probing algorithms, because these algorithms are in some sense subsumed by the Symmetric algorithm. Figure 2 represents the state diagram for the Forward probing algorithm operating at a single node using an arbitrary threshold T. The state of the node is represented by a tuple (N_t, J_t), where N_t is the number of tasks at a node and J_t is the probe state that indicates if the node is being forward probed or not. The probe states have the following codes:

- 0 : if not being probed,
- 1 : if being forward probed,

The infinitesimal generator matrix corresponding to this process is:

$$Q_F = \begin{Vmatrix} & B_{01} & \dots & B_{01} & B_{01} & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & A_1 & A_1 & A_1 & \dots \\ A_2 & A_2 & \dots & A_2 & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

with exactly $T-1$ columns of (B_{01}, B_{11}, A_2).

Figure 3 represents the state diagram for the Reverse probing algorithm operating at a single node using an arbitrary threshold T. The state of the node is represented by a tuple (N_t, J_t), where N_t is the number of tasks at a node and J_t is the probe state that indicates if the node is either probing or not. The probe states have the following codes:

- 0 : if not probing,
- 1 : if reverse probing,

The infinitesimal generator matrix for this process is:

$$Q_R = \begin{Vmatrix} & A_0 & \dots & A_0 & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & B_{11} & A_1 & A_1 & \dots \\ B_{10} & B_{10} & \dots & B_{10} & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

with exactly $T - 1$ columns of (A_0, B_{11}, B_{10}).

All the parameters for the Forward and Reverse probing algorithms have the same meanings as their counterparts in the Symmetric algorithm. For instance, μ' in Reverse probing is the rate at which a node sends out tasks in response to reverse probes made by other nodes, as in the case of Symmetric probing.

The computational procedure for both these algorithms is very similar to that for the Symmetric probing algorithm, which was described earlier in this Section in detail. The model is solved iteratively in both cases. The unknown parameters in the case of Forward are α and h and in the case of Reverse, the unknowns are μ' and q. The internals of the matrices of Forward and Reverse are described in Appendix B.

In both of these cases, initial values of the unknown parameters are used to solve the model. Based upon this solution, new values of the parameters are determined. The iteration continues until the stopping criterion has been satisfied. It was seen that the iteration was insensitive to the initial values chosen for the unknown parameters. Further, the number of iterations was usually small, between 10 and 20.

4 Performance Comparisons

In this section, the performance of the three load sharing algorithms will be compared to each other and to two bounds, represented by the no-load-balancing $M/M/1$ model (also referred to as NLB) for K nodes and the perfect load sharing with zero costs, i.e., the $M/M/K$ model. Wherever relevant, we will also compare the algorithms against a Random assignment algorithm, which transfers tasks based only upon local state information. This algorithm is similar to Forward in the sense that a node that goes above $T + 1$ transfers a task. However, the node does not send any probes. Instead, it picks a destination node at random and transfers a task to this node. The key performance metric for comparison is the mean response time of tasks.

A large number of parameters such as the service time, the threshold T, the probe limit L_p, the communication delay $1/\gamma$, the number of nodes in the network etc., can affect the performance of load sharing algorithms. In this connection, we will try to present the results that we believe are the most relevant. The presentation will be in the following sequence:

- Validation of the analytical results with simulations.
- Nominal comparisons between the algorithms.
- Relation between delays and thresholds.
- Optimal response times as a function of delays.
- Optimal thresholds as a function of delays.

Unless specifically mentioned otherwise, $L_p = 2$ in all the runs. Also, $S = 1/\mu$ and $C = 1/\gamma$ are the means of the service time and task transfer delay, respectively. Further, it will be assumed that $S = 1$ unit and all measurements of response times will be in terms of this unit.

Validation with Simulations

We mentioned in Section 2 that the decomposition used in this paper is only an approximate solution which is conjectured to be exact for infinitely large systems. Thus, it is important to determine how well this approximation compares to simulations of finite sized systems. The simulation model consisted of 10 nodes in all cases except when $\rho = 0.9$, where the model consisted of 20 nodes. Figure 4 depicts a representative set of curves regarding this study.

Because the simulation results were almost identical to the analytical model, we have chosen not to depict the actual sample means of the response times from the simulations. Instead, the 95% confidence intervals of the simulation results are presented, as computed by the Student-t tests. On the average, the confidence interval for the response time is about ±0.015 units about the sample mean. The only exception to this is at $\rho = 0.9$, when the confidence interval is about ±0.165 units about the sample mean.

We have observed (results not presented here) that in most of the cases, the variation between the simulation results and the analytical models is less than 2%. Furthermore, the model is almost invariably optimistic, compared to the simulation results. The maximum variation that we observed was about 15%, and such numbers were very infrequent and were seen to occur at low communication delays and high loads ($\rho \geq 0.9$). As the delays increase however, the model tends to become more accurate. In any case, for loads ≤ 0.8, the model is a very good approximation, even for reasonably small systems. In cases where the variation was more than 2%, it was seen that by increasing the size of the simulation system to 20 nodes, the results generated better agreement with those of the analytical model. For instance, the variation at $\rho = 0.9, C = 0.1S$, which was about 15%, decreases to about 5% for a system of 20 nodes.

Comparison of the Algorithms

In an earlier study by Wang and Morris[WANG85], it was postulated that at low traffic intensities, Forward probing is likely to perform best, while at high traffic intensities, Reverse would be more suitable. However, it was not known exactly where one policy became better than the others, especially when there are significant communication delays involved.

Another factor that takes on a degree of importance in this comparison between algorithms, is that of probe overhead. While we have assumed that probes take zero time, there is the potential for the probes to interfere with other messages, especially if they are generated in large enough numbers. It has been shown in [MIRC87] that the Symmetric algorithm generates probes at a higher rate than do Forward and Reverse. While we have not included the effects of such overhead in our model thus far, this aspect of the study is currently under progress.

Figure 5 shows the performance curves of the algorithms for $C = 0.1S$ and $T = 0$. From this figure, we can make the following observations:

- At low delays and low loads ($\rho \leq 0.5$), Forward performs essentially like Symmetric but Reverse is worse by as much as 30%. This can be explained by the fact that in most cases, Reverse is ineffective in load sharing as most nodes will not have a spare task. Thus, the Reverse component of Symmetric does not improve its performance over Forward.
- At moderate loads, Symmetric performs much better than both Forward and Reverse, by as much as 20%, while Forward and Reverse are about the same.
- At high loads ($\rho > 0.9$), it is seen that Reverse is better than Forward by a substantial margin of about 25% while Symmetric is still the best overall, being better than Reverse by about 25%.
- At all the loads tested, there appears to be a substantial gain in load sharing as opposed to NLB. This is true for all three algorithms. However, the improvement is much more pronounced as the load increases. For instance, at $\rho = 0.9$, the response time for Symmetric is about 2 units whereas the NLB response time is 10 units, a significant difference.

- As may be expected, the algorithms perform worse than the exact $M/M/K$ model. However, Symmetric generates close performance to the $M/M/K$ model. For instance, at $\rho = 0.9$, $M/M/K$ results in a response time of 1.3 units while Symmetric generates 2 units.

Figure 6 shows the performance curves of the algorithms for $C = 2S$ and $(T = 2)$. From Figure 6, we can reach the following conclusions:

- For moderate communication delays and low to moderate loads ($\rho \leq 0.7$), the behavior of the three algorithms is virtually the same. It would appear that the delay overhead predominates at these loads.
- At moderate loads, ($\rho = 0.8$), Symmetric is about 10% better than Reverse but almost identical to Forward.
- Only at very high loads ($\rho \geq 0.9$) does Symmetric actually perform significantly better than both Forward and Reverse.
- In comparison with NLB, it is seen that at low loads ($\rho \leq 0.5$), there is little if no improvement by load sharing. However, as the load increases, load sharing becomes more viable. At $\rho = 0.9$, Symmetric generates a response time of 3.5 units as opposed to 10 units for NLB.
- The comparison against the $M/M/K$ model is not very flattering at high delays, as might be expected. For instance, Symmetric at $\rho = 0.9$ is about 2.5 times worse than the $M/M/K$ value of about 1.3 units.

Thus, one can conclude that at moderately high delays, the performance of the three algorithms is virtually identical. A surprising result though is that Symmetric is significantly better at very high loads.

All the subsequent discussion is based on the results obtained from the Symmetric algorithm. Unless explicitly mentioned otherwise, the conclusions reached are also applicable to Forward and Reverse. in cases where the performance of these algorithms is markedly different from that of Symmetric, a separate discussion will be provided.

Delays vs Thresholds

Figures 7, 8 and 9 show the response times for the Symmetric algorithm tested over a wide range of communication delays and thresholds, for the traffic intensities of 0.5, 0.7 and 0.9. It can be seen from Figure 7, that at low delays ($C = 0.1S$), the optimal threshold is 0 and the performance is a monotonically increasing function of the threshold. Also, the response time generated at $T = 0$ is only about 20% worse than the exact $M/M/K$ value for moderate loads ($\rho \leq 0.7$). For example, at $\rho = 0.7$, the Symmetric response time is about 1.3 units whereas the exact $M/M/K$ value is approximately 1.04 units. Further, the NLB resonse time for this load is 3.3 units, which is much worse than the performance of the Symmetric algorithm. The performance improvement due to load sharing in this case can be explained by the following arguments:

- At low delays, the cost of transferring a task is much lower than the potential improvement due to the effect of load sharing. Thus, $T = 0$ permits very active load sharing.
- Because the delays are small, much greater certainty exists in the knowledge that an idle node will continue to remain idle during the time it takes to transfer a task to it. Thus, in some sense, $T = 0$ ensures that all task transfers are useful in that a remote task arrives at the node soon after it becomes idle.

For moderate delays ($C = S$, Figure 8), the behavior is as follows: Even at $\rho = 0.5$, there is a gain of about 22% from load sharing. For instance, the best response time at this load is about 1.56 units while the corresponding NLB performance is 2 units. The improvements over NLB by load sharing at higher loads are even more substantial, being as high as about 73% for $\rho = 0.9$. The NLB response time in this case is 10 units whereas the optimal Symmetric value is about 2.7 units, as can be seen from Figure 8. Further, $T = 1$ for $\rho = 0.5$ and 0.7, while $T = 2$ for $\rho = 0.9$, are the ***optimal*** thresholds.

When the communication delays increase to the order of $10S$ (Figure 9), it is seen that the best that can be achieved for $\rho = 0.5$ is the NLB performance which is 2.0 units response time. Thus, it would be appropriate to turn off load sharing here. For $\rho = 0.7$, a small gain of about 5% is seen, at $T = 5$. This impovement is small enough that if the interference of probes could be accounted for, the best strategy might very likely be to turn off load sharing. However, at $\rho = 0.9$, the reduction in response time from the NLB is about 40% and this occurs at $T = 6$, where the

Symmetric response time is about 6.0 units. In any case, the response times at high delays are significantly worse than the $M/M/K$ values as might be expected. For instance, at $\rho = 0.7$, the $M/M/K$ response time is 1.04 units whereas the best load sharing value is about 3.1 units.

Optimal Response Times

The purpose of this set of tests is to determine the best possible performance of the algorithms under a very large range of transfer delays, ranging from as small as $1/100\,S$, to as large as $100\,S$. Thus, in this study, one can assume very fast local area networks will form one end of the spectrum and slow, long-haul networks the other end. Figure 10 shows the results of the tests for the algorithm.

The response time in each case is normalized by the $M/M/1$ response times. Thus, a lower ratio indicates greater improvements as a consequence of load sharing. Corresponding to each curve representing a particular traffic intensity, there is a curve for the performance of the Random assignment algorithm, to be used as a baseline. From the figure, one can see that at low delays ($\leq 1/2\,S$), the gain from load sharing is quite substantial, at all traffic intensities considered. Further, the gains are greater for higher loads. At loads of 0.9, the response times are 0.25 times those for the no load sharing case.

As can be seen from the curves representing the performance of the random assignment, there is a definite advantage in probing. However, as the delays increase, ($> 1\,S$), this advantage of probing seems to disappear. Random with a suitable threshold is able to perform as well as any probing policy, giving the impression that the state information due to probing is so out of date as to not really be useful. Also, the best that can be achieved in lower traffic intensities (≤ 0.5) is no better than the $M/M/1$ response time at these delays. However, there is still a marked improvement in the performance of load sharing at higher loads, for example at 0.8 and 0.9. The remarkable fact that should be noticed here is that even at delays as high as $100\,S$, there is about an 8% improvement over no load sharing for traffic intensity 0.9. We postulate that at higher loads, this effect will be even more prominent.

Optimal thresholds

Figure 11 indicates the variation of the *optimal* thresholds corresponding to the *optimal* response times indicated in Figure 10. Note that the thresholds are low at lower delays and get higher as the delays increase. Further, this effect is seen to be more prominent at higher traffic intensities. At $\rho = 0.9$, the optimal threshold varies between 0 when the delay is $1/10\,S$ and 25, when the delay is $100\,S$. The variation is significantly lower at low loads.

5 Summary and Conclusions

This study was concerned with the performance analysis of simple load sharing algorithms in the presence of significant task transfer delays. The three algorithms that we tested were called Forward, Reverse and Symmetric. The analysis of the algorithms was carried out using the Matrix-Geometric solution technique.

The Markov process of the entire network appeared to be computationally intractable. Thus, we employed a decomposition technique to solve the Markov process. While this resulted in an approximate solution of the original system, it was seen by means of simulation studies, that the variation between the exact and approximate solutions was minimal for systems of 10-20 nodes. Consequently, the analytical solution is likely to be more accurate for larger systems. This leads us to hypothesize that the decomposition is an exact solution of the system in the limit as the number of nodes tends to infinity.

The three load sharing algorithms were tested over a large range of parameter values. Some of the salient observations that we made were as follows:

- There is considerable difference between the performance of the three algorithms at low to moderate delays ($\leq S$), with Symmetric providing the best results. As delays increase, the algorithms tend to provide almost identical performance, especially when ($D \geq 10S$). Further, at such delays, Random assignment performs as well as any probing scheme, leading us to believe that at moderate to high delays, probing is wasted effort.
- At high delays ($\geq 10S$), the optimal response times are no better than those for the NLB case, leading us to believe that load sharing is not useful in such situations, for low to moderate loads. However, at high loads ($\rho \geq 0.9$), susbtantial benefits accrue from load sharing even at these delays.
- Reverse probing is outperformed by Forward over most of the range of loads tested, except when $\rho \geq 0.9$. While Symmetric is the best of the three algorithms tested, it does have the potential for generating high probing over-

heads. Given these observations, Forward would appear to have even greater applicability if realistic overhead costs might be assigned to probes.

- The benefits of load sharing are more pronounced at high loads ($\rho \geq 0.8$). This is evidenced by the fact that the percentage reduction in response times in these cases is greatest over the corresponding NLB values.
- At extremely high loads $\rho = 0.9$, it is seen that about 8% reduction is achieved over the corresponding NLB response time, even when the delays are as high as $100S$.
- The optimal threshold was seen to be a function of the load and the task transfer delay. At low delays, the optimal threshold was 0 for all the loads tested. However, as the delays increased, the optimal threshold increased correspondingly, becoming about 24 for $\rho = 0.9$ and delay $= 100S$.

Appendix A

In this appendix, we give closed form representations of the matrices A_0, A_1, A_2 and the matrices $B_{00}, B_{01}, B_{10}, B_{11}$, and B_{21}, for the Symmetric probing algorithm.

$$B_{00} = \begin{bmatrix} -(\alpha+\lambda) & 0 & \alpha & 0 \\ 0 & -(\alpha+\gamma+\lambda) & 0 & \alpha \\ 0 & 0 & -(\gamma+\lambda) & 0 \\ 0 & 0 & 0 & -(2\gamma+\lambda) \end{bmatrix}$$

$$B_{01} = \begin{bmatrix} \lambda & 0 & 0 & 0 \\ \gamma & \lambda & 0 & 0 \\ \gamma & 0 & \lambda & 0 \\ 0 & \gamma & \gamma & \lambda \end{bmatrix}$$

$$B_{10} = \begin{bmatrix} \mu q & \mu\bar{q} & 0 & 0 \\ 0 & \mu & 0 & 0 \\ 0 & 0 & \mu q & \mu\bar{q} \\ 0 & 0 & 0 & \mu \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} -\sigma & 0 & 0 & 0 \\ 0 & -(\gamma+\sigma) & 0 & 0 \\ 0 & 0 & -(\gamma+\sigma) & 0 \\ 0 & 0 & 0 & -(2\gamma+\sigma) \end{bmatrix}$$

$$A_0 = \begin{bmatrix} \lambda h & 0 & 0 & 0 \\ \gamma & \lambda h & 0 & 0 \\ \gamma & 0 & \lambda h & 0 \\ 0 & \gamma & \gamma & \lambda h \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -\delta & 0 & 0 & 0 \\ 0 & -(\gamma+\delta) & 0 & 0 \\ 0 & 0 & -(\gamma+\delta) & 0 \\ 0 & 0 & 0 & -(2\gamma+\delta) \end{bmatrix}$$

$$A_2 = (\mu + \mu')I_4$$

where

$$\delta = (\lambda h + \mu + \mu'),$$
$$\sigma = (\lambda + \mu),$$

and I_4 is the identity matrix of size 4.

Appendix B

In this appendix, we provide closed form representations for the matrices in the case of the Forward and Reverse probing algorithms.

Forward

$$B_{00} = \begin{bmatrix} -(\alpha+\lambda) & \alpha \\ 0 & -(\gamma+\lambda) \end{bmatrix}$$

$$B_{01} = \begin{bmatrix} \lambda & 0 \\ \gamma & \lambda \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} -(\alpha + \lambda + \mu) & \alpha \\ 0 & -(\mu + \gamma + \lambda) \end{bmatrix}$$

$$x = \sum_{i \leq T} \mathbf{p}_i \, [01]^T + \sum_{i > T} \mathbf{p}_i e$$

which is the probability that a node will respond negatively to a forward probe. Thus, $\bar{x} = 1 - x$ is the probability that a node will respond positively to a forward probe.

If a node probes L_p nodes, then the probability that the set of probes results in failure is

$$h = x^{L_p}$$

$$A_0 = \begin{bmatrix} \lambda h & 0 \\ \gamma & \lambda h \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -(\mu + \lambda h) & 0 \\ 0 & -(\mu + \lambda h + \gamma) \end{bmatrix}$$

$$A_2 = \mu I_2$$

where I_2 is the identity matrix of size 2.

Also, $R = [r_{i,j}]$ can be written as follows:

$$\begin{aligned} r_{1,1} &= \lambda h / \mu \\ r_{2,2} &= \frac{\theta + \gamma - ((\theta + \gamma)^2 - 4\mu\lambda h)^{1/2}}{2\mu} \\ r_{1,2} &= 0 \\ r_{2,1} &= \frac{\gamma}{\theta - (r_{1,1} + r_{2,2})\mu} \end{aligned}$$

where $\theta = \lambda h + \mu$. It can be shown that the stability criterion for the Forward probing algorithm is

$$\lambda h < \mu.$$

Reverse

To determine q, the probability of a set of reverse probes resulting in failure, we use the following procedure:

Let

$$y = \sum_{i \le T+1} \mathbf{p}_i e$$

If the node probes L_p nodes to receive a remote task, then the probability that all of them will be unsuccessful is denoted by: $q = y^{L_p}$, and $\bar{q} = 1 - q$ is the probability that at least one of the reverse probes is successful.

$$B_{00} = \begin{bmatrix} -\lambda & 0 \\ 0 & -(\lambda + \gamma) \end{bmatrix}$$

$$B_{10} = \begin{bmatrix} \mu q & \mu \bar{q} \\ 0 & \mu \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} -(\mu + \lambda) & 0 \\ 0 & -(\mu + \gamma + \lambda) \end{bmatrix}$$

$$A_0 = \begin{bmatrix} \lambda & 0 \\ \gamma & \lambda \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -(\mu' + \lambda) & 0 \\ 0 & -(\mu' + \lambda + \gamma) \end{bmatrix}$$

$$A_2 = (\mu + \mu')I_2$$

Also, $R = [r_{i,j}]$ can be written as follows:

$$\begin{aligned} r_{1,1} &= \lambda/(\mu+\mu') \\ r_{2,2} &= \frac{\phi+\gamma-((\phi+\gamma)^2-4(\mu+\mu')\lambda)^{1/2}}{2(\mu+\mu')} \\ r_{1,2} &= 0 \\ r_{2,1} &= \frac{\gamma}{\phi-(r_{1,1}+r_{2,2})(\mu+\mu')} \end{aligned}$$

where $\phi = \lambda + \mu + \mu'$. It can be shown that the stability criterion for the Reverse probing algorithm is

$$\lambda < \mu + \mu'.$$

Acknowledgements The authors would like to thank the referees for their valuable comments. In particular, one referee pointed out the flaw in our initial analysis pertaining to the reducibility of A and suggested the correction that appears in this paper.

References

[BOKH79] Bokhari, S., "Dual Processor Scheduling With Dynamic Reassignment," *IEEE Trans. Soft. Engg.*, Vol. SE-5, No. 4, July 1979.

[dSeS84] de Souza e Silva, E. and M. Gerla, "Load Balancing in Distributed Systems With Multiple Classes and Site Constraints," *Performance '84*, , pp. 17–33, 1984.

[EAGE86] Eager, D., E. Lazowska, and J.Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Soft. Engg.*, Vol. SE-12, No. 5, pp. 662–675, May 1986.

[LATO81] Latouche, G., "Algorithmic Analysis of a Multiprogramming-Multiprocessor Computer System," *J. ACM*, Vol. 28, October 1981.

[LEE87] Lee, K. J., *Load Balancing in Distributed Computer Systems*, PhD thesis, ECE Dept., University of Massachusetts, February 1987.

[LIVN82] Livny, M. and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Performance Evaluation Review*, Vol. 11, No. 1, pp. 47–55, 1982.

[MIRC87] Mirchandaney, R., L. Sha, and J. A. Stankovic, "Load Sharing in the Presence of Non-Negligible Delays," 1987. in preparation.

[NEUT81] Neuts, M. F., *Matrix-Geometric solutions in Stochastic Models: An Algorithmic Approach, Mathematical Sciences*, Johns Hopkins University Press, 1981.

[STAN85] Stankovic, J., "Bayesian Decision Theory and Its Application to Decentralized Control of Task Scheduling," *IEEE Trans. Computers*, Vol. C-34, No. 2, pp. 117–130, February 1985.

[STON78a] Stone, H., "Critical Load Factors in Two Processor Distributed Systems," *IEEE Trans. Soft. Engg.*, Vol. SE-4, No. 3, May 1978.

[STON78b] Stone, H., "Multiprocessor Scheduling with the Aid of Network Flow algorithms," *IEEE Trans. Soft. Engg.*, Vol. SE-3, No. 1, May 1978.

[TANT85] Tantawi, A. and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *J. ACM*, Vol. 32, pp. 445–465, Apr. 1985.

[THEI85] Theimer, M., K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings of the 10th Symposium on Operating System Principles*, December 1985.

[TOWS86] Towsley, D., "The Allocation of Programs Containing Loops and Branches on a Multiple Processor System," *IEEE Trans. Soft. Engg.*, Vol. SE-12, pp. 1018–1024, October 1986.

[WANG85] Wang, Y. and R. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Computers*, Vol. C-34, March 1985.

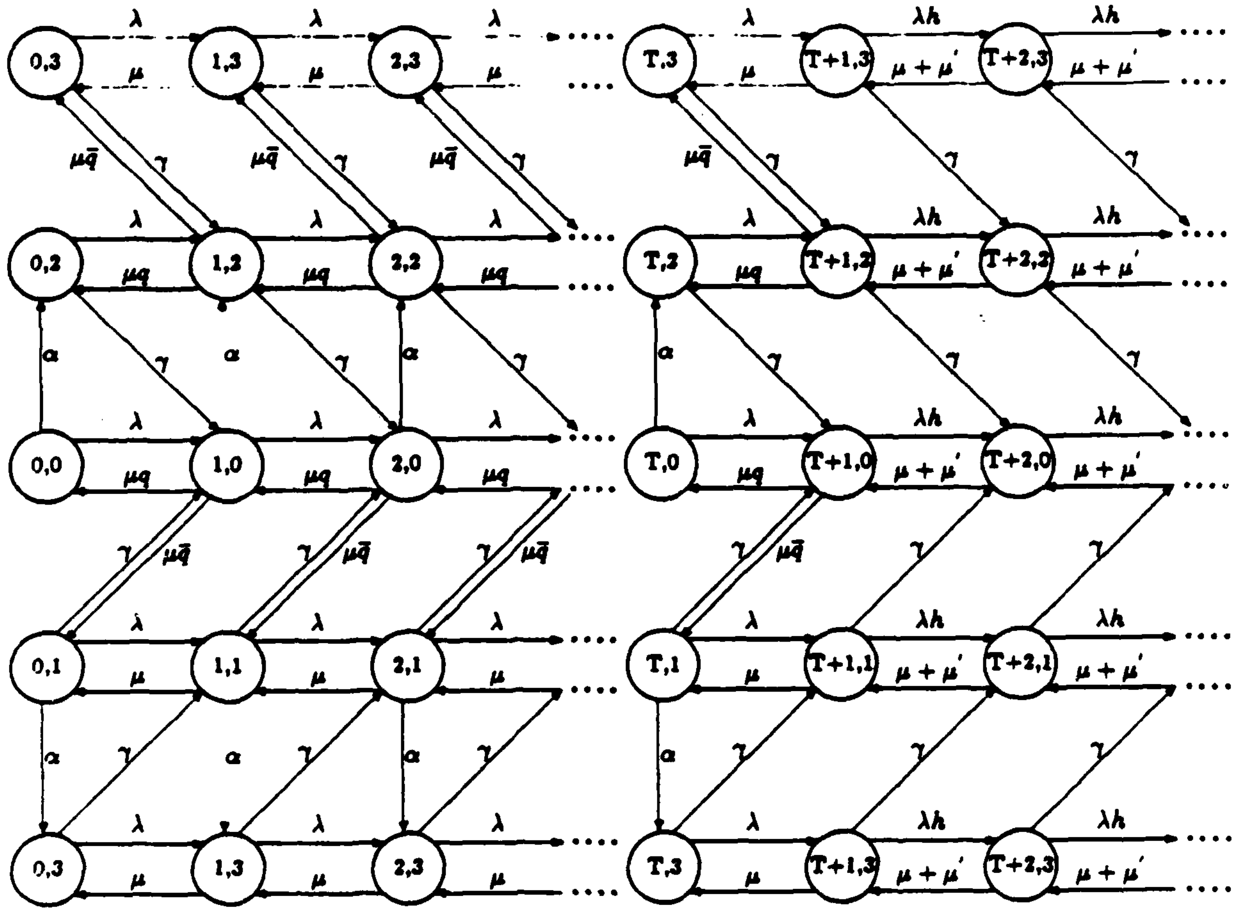


Figure 1. Symmetric Probing

]

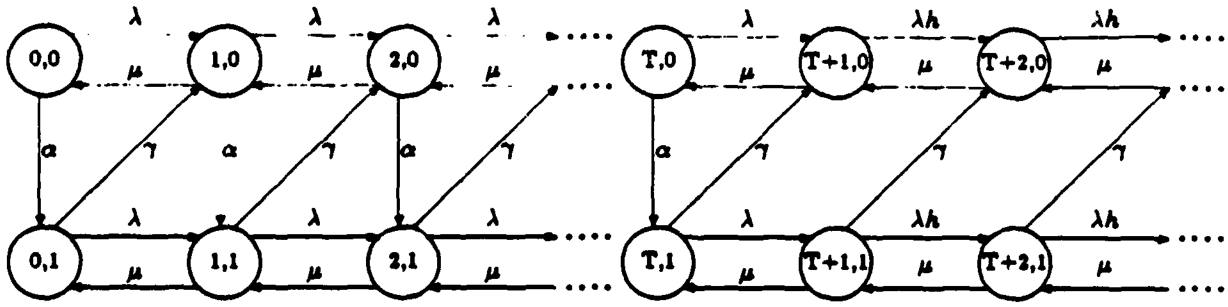


Figure 2. Forward Probing

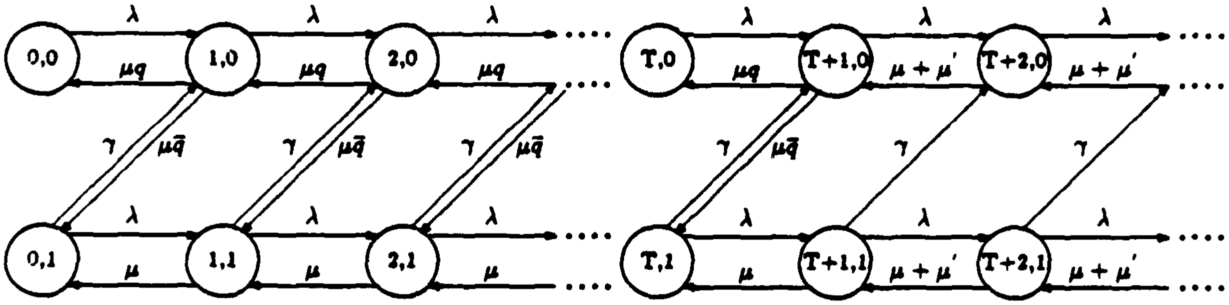
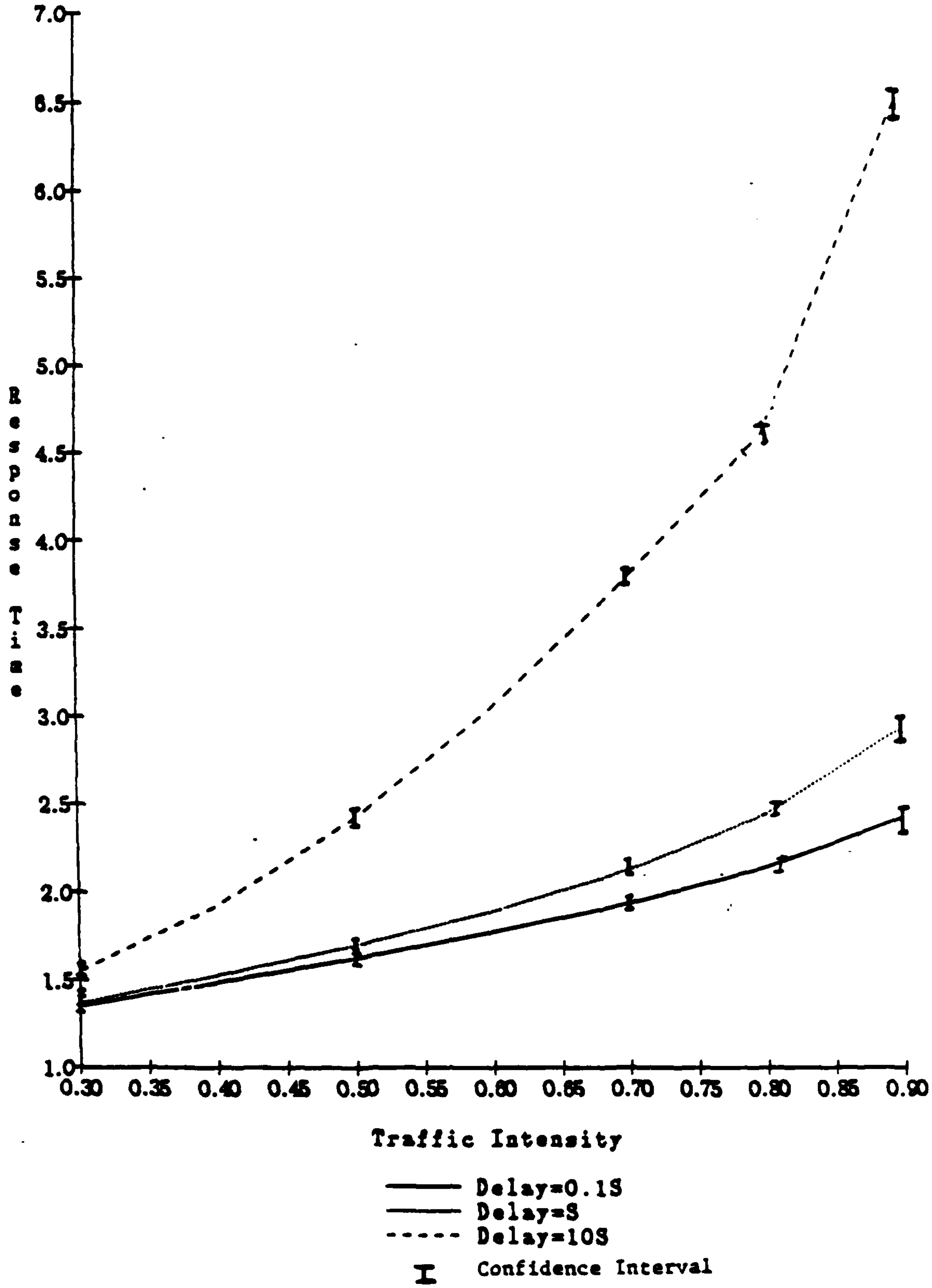


Figure 3. Reverse Probing

Figure 4. Validation with Simulations



TS1

Figure 5. Comparison of Algorithms

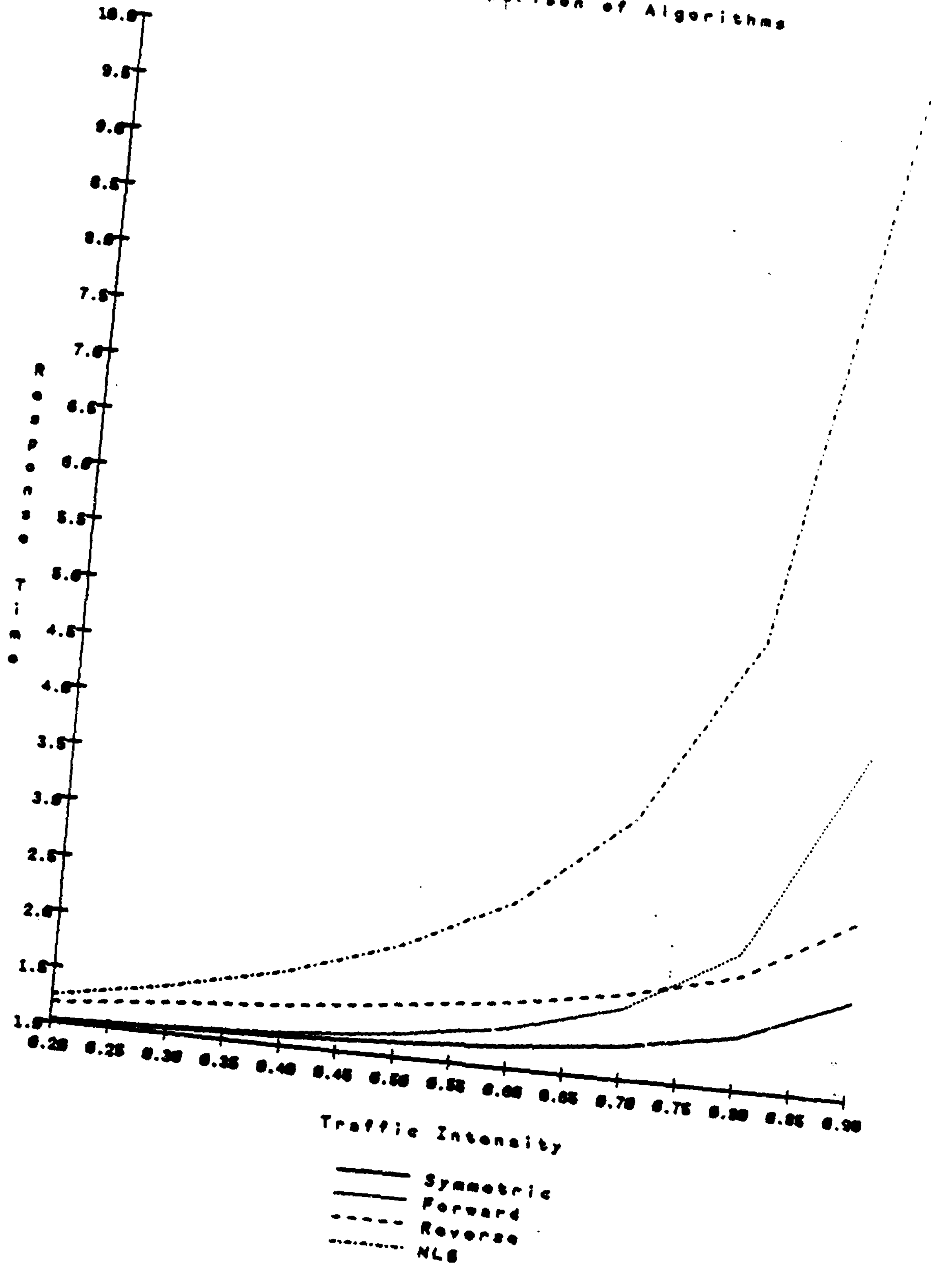
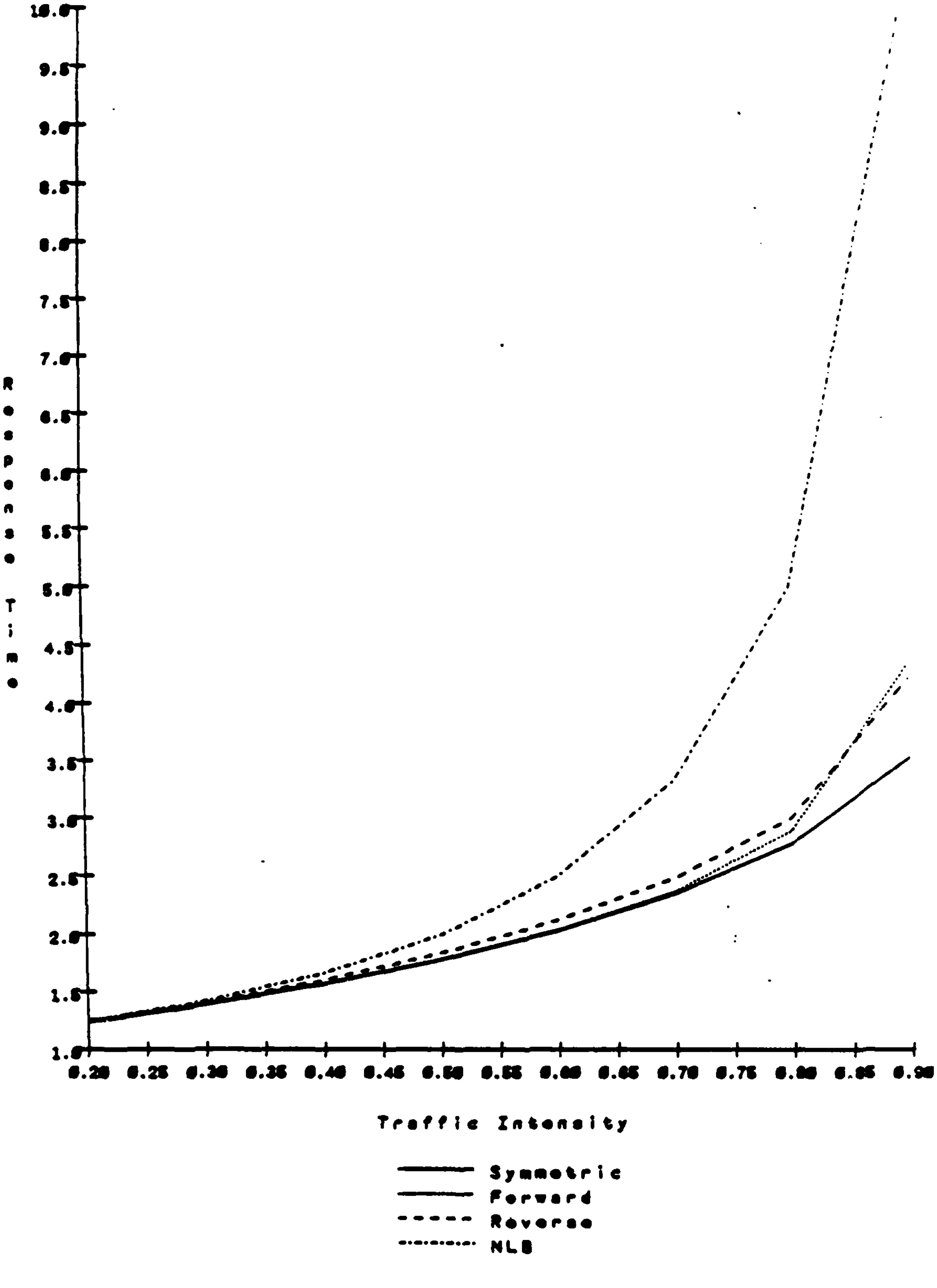


Figure 6. Comparison of Algorithms



Figure·7.·Variation·of·Threshold·(Delay=0.1S)

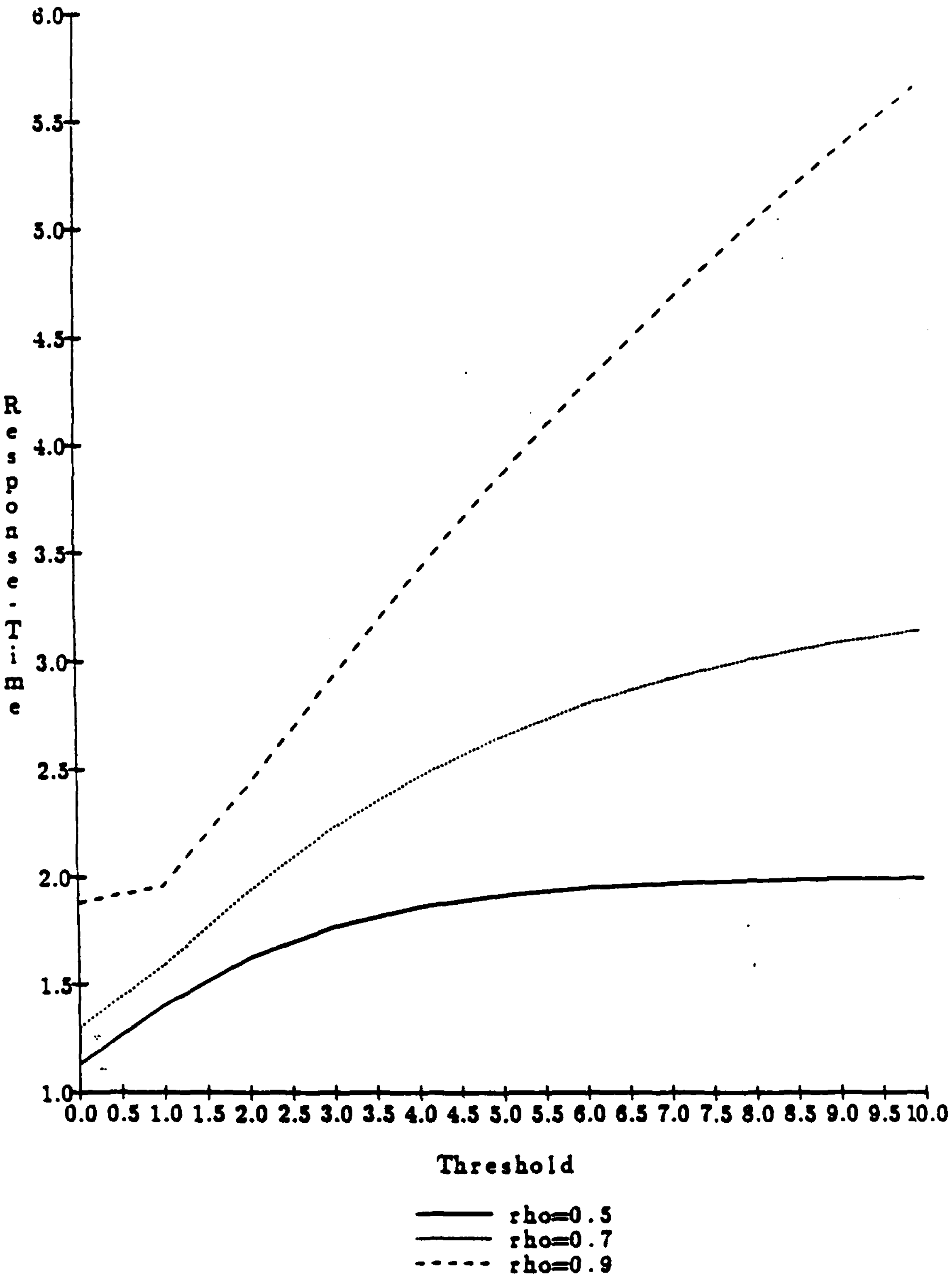


Figure-8.-Variation-of-Threshold-(Delay=S)

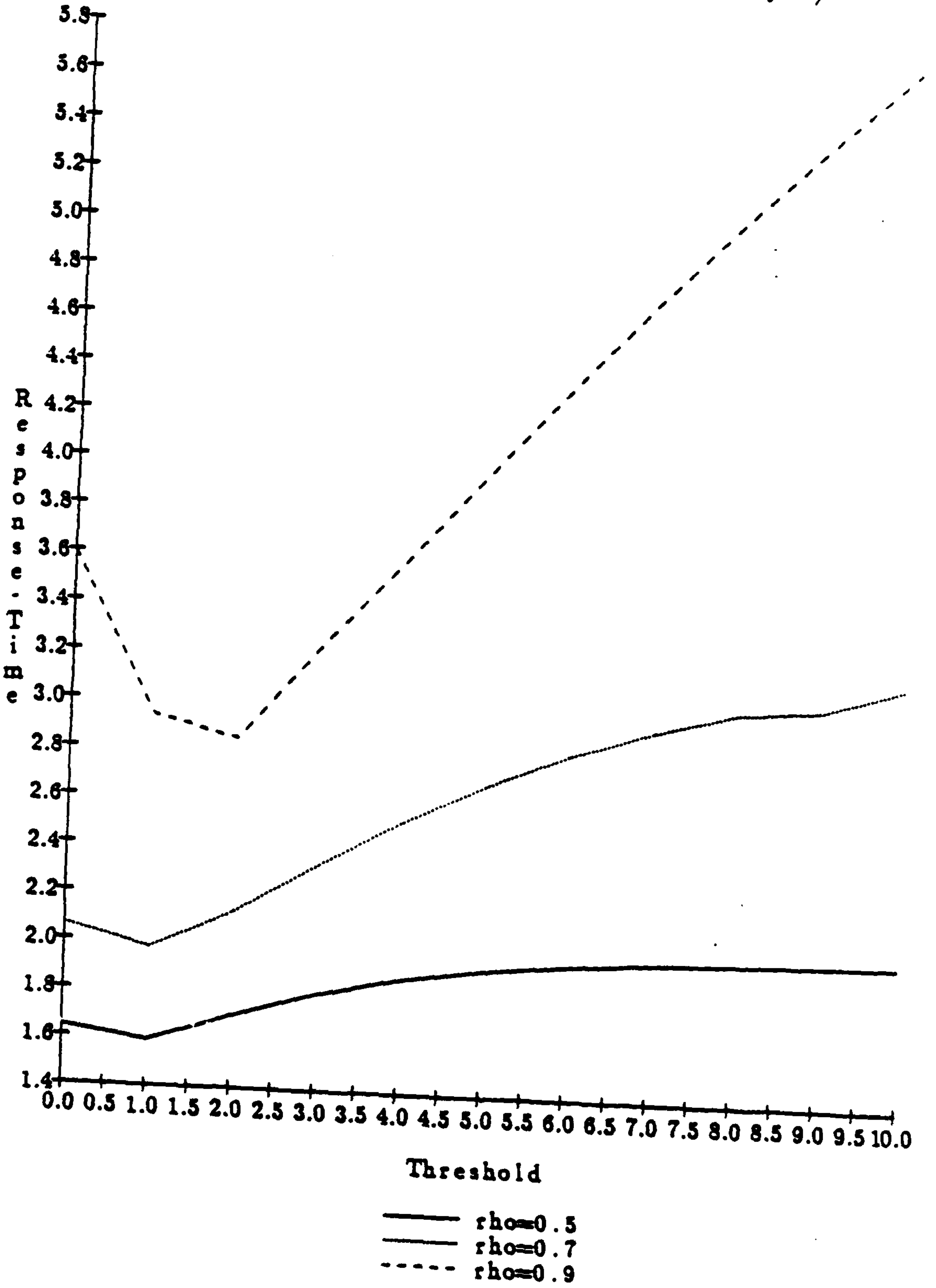
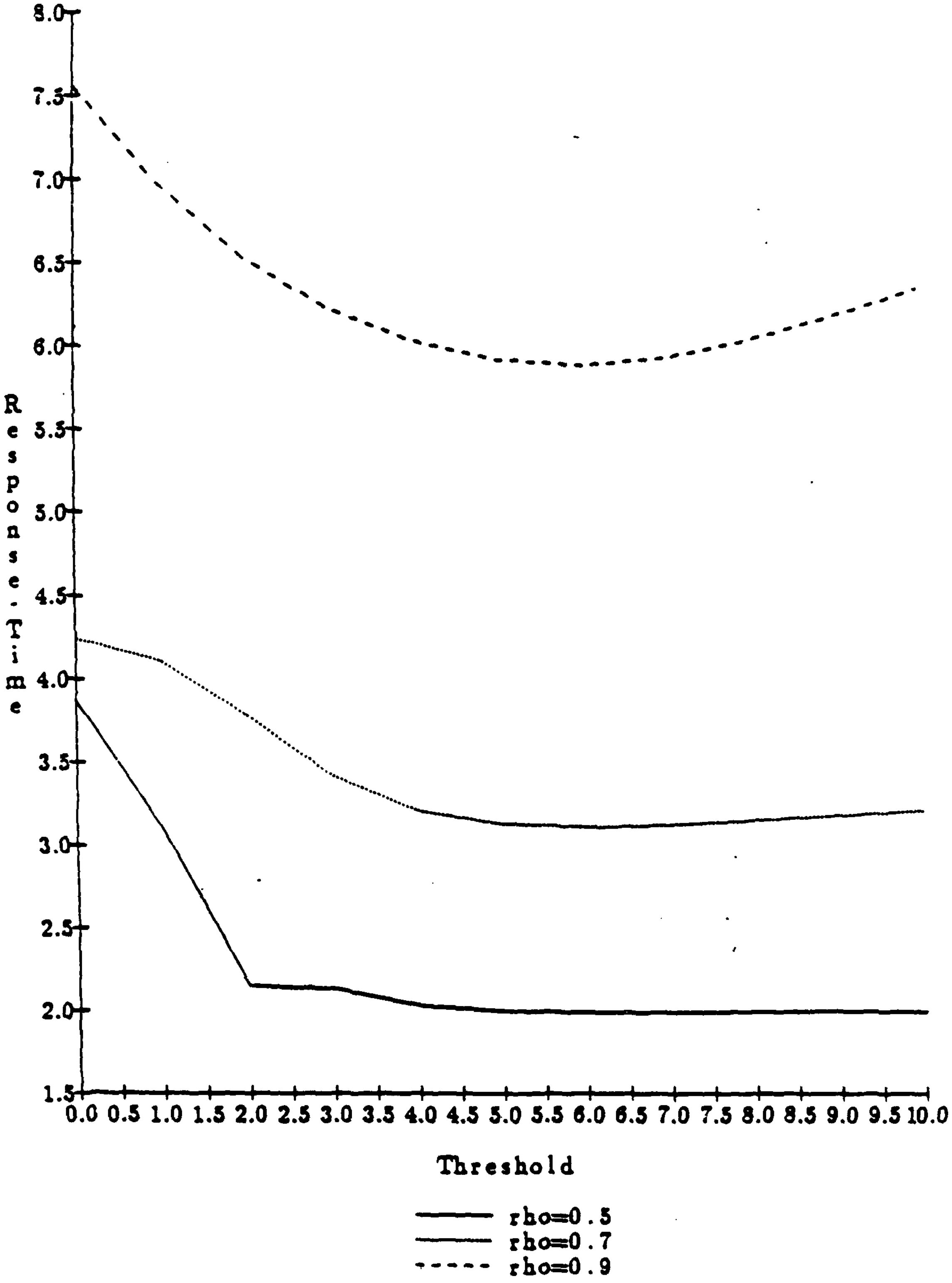
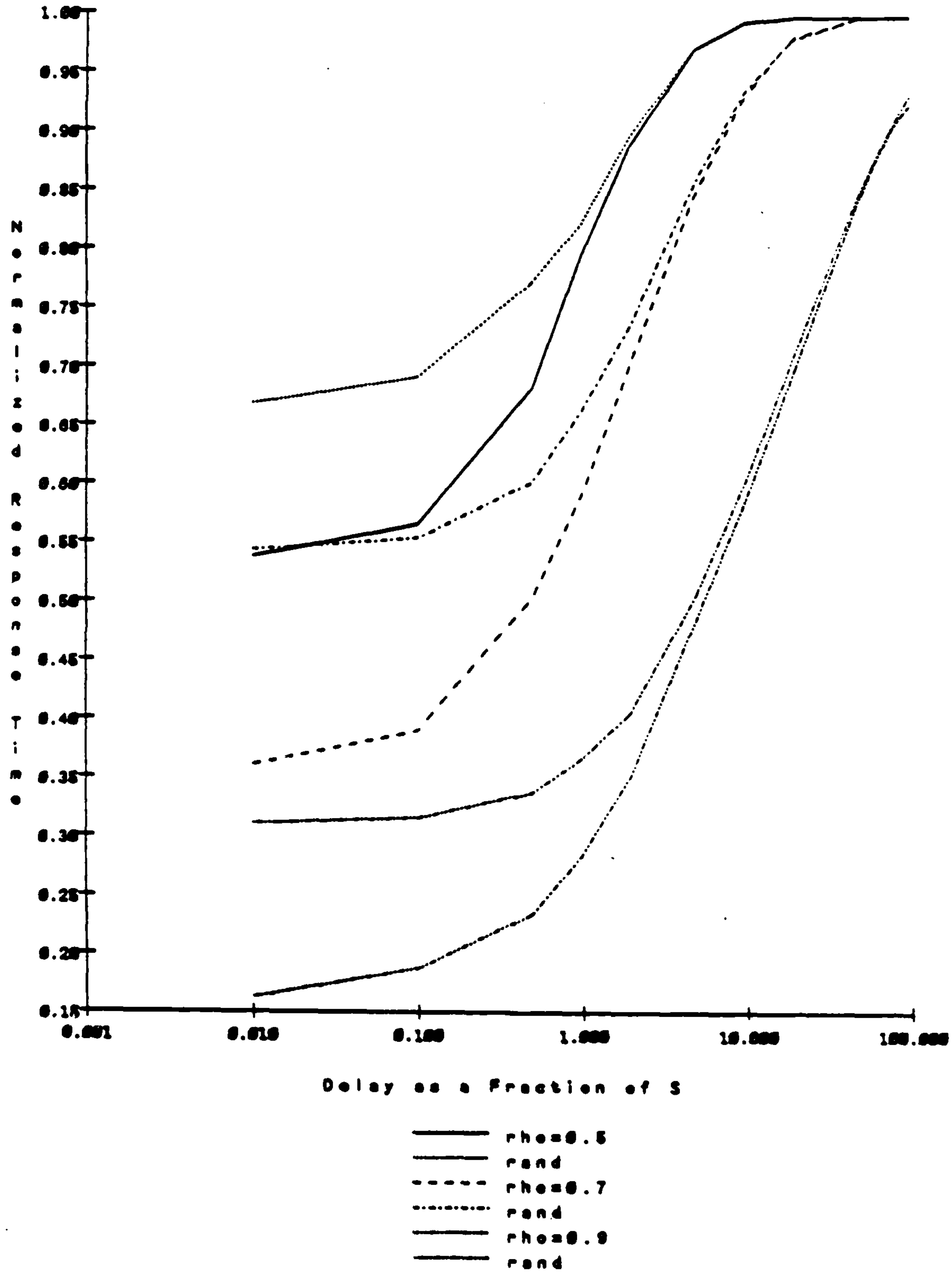


Figure 9. Variation of Threshold (Delay=10S)



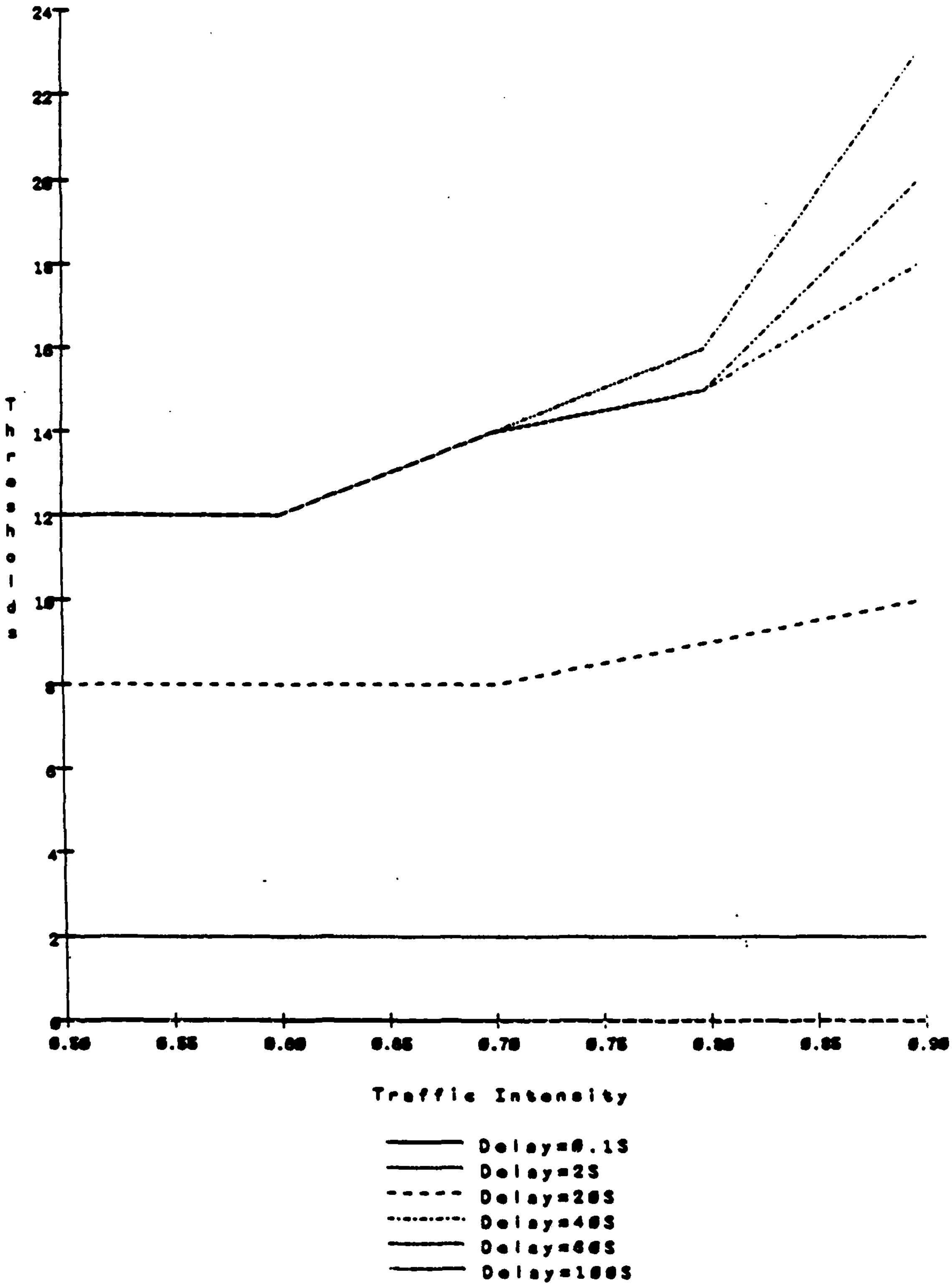
T01

Figure 10. Optimal Normalized Response Times



T01

Figure 11. Variation of Thresholds



Effects of Delays on Receiver-Initiated Load Sharing Policies *

Ravi Mirchandaney
ECE Dept., U.Massachusetts
Amherst MA. 01003

Don Towsley
COINS Dept., U.Massachusetts
Amherst MA. 01003

John A. Stankovic
COINS Dept., U.Massachusetts
Amherst MA. 01003

March 1987

Abstract

In this paper, we study the performance characteristics of simple Receiver-Initiated load sharing algorithms for distributed systems. There have been various studies which indicate that under certain assumptions, non-negligible delays are encountered in transferring tasks from one node to another. Further, the state information that is gathered by the load sharing algorithms is out of date by the time the load sharing decisions are taken. This paper analyzes the effects of these delays on the performance of Receiver-Initiated algorithms that we call R_K, and R_{K_T} and variations of these algorithms R_D and R_{D_T}. We also study the effectiveness of a dual-threshold algorithm called R_{T^2}. Approximate queueing models are developed for each of the algorithms operating in a homogeneous system under the assumption that the task arrival process at each node is Poisson and the service times and task transfer times are exponentially distributed. The models are solved using the Matrix-Geometric solution technique. Some of the interesting observations that we have made are as follows:

*This work was supported, in part, by the National Science Foundation under grant ECS-8406402 and by RADC under contract RI-44896X.

The analytical model is shown be a very good approximation of the underlying system. It is seen that the algorithms are insensitive to the parameter K and the effects of probing delays are determined to be negligible, under reasonable assumptions regarding probe sizes.

1 Introduction

One potential advantage of distributed systems is to share computation power between nodes. This is referred to as load balancing. Distributed load balancing has been an active area of research for some time. The approaches used to investigate this problem have been quite diverse and includes simulation studies, analytical studies and actual implementations of load sharing algorithms on real systems.

Within the context of simulation studies, some of the interesting work has been as follows: Adaptive load sharing has been studied by Livny and Melman[LIVN82], who showed that in a network of nodes, there is a very high probability that the queues are unbalanced. They also develop a taxonomy of load sharing policies and evaluate them by simulation. Bryant and Finkel[BRYA81] performed simulation studies of an adaptive load sharing policy. Stankovic[STAN84] performed simulation studies of simple load balancing algorithms, and there have been others.

Under certain simplifying assumptions, it has been possible to perform formal analyses of load sharing algorithms. As regards the analytical approach, Stone[STON78b] and[STON78a], Bokhari[BOKH79] and Towsley[TOWS86] examined static algorithms that utilized information about the average behavior of tasks in deciding their assignments. Tantawi and Towsley[TANT85] investigated an optimal probabilistic assignment scheme. Silva and Gerla[dSeS84] determine an optimal load sharing strategy under the assumption that the nodes and the communication network can be modelled as product form queueing networks. Recently, Lee[LEE87] studied the effects of task transfer delays on simple algorithms that do not utilize any remote state information. Eager et.al.[EAGE86] evaluated three simple adaptive load sharing schemes. In this work, they assume that the entire overhead due to load sharing is transferred onto the CPU and is modelled as an increased load on the same. Further, the nodes are assumed to be part of a local area network connected by a high bandwidth medium. Thus, there are no communication delays for task transfers and remote state information is always perfectly accurate.

Finally, the proliferation of distributed systems has given rise to actual implementations of simple load sharing algorithms, for example, in the Stanford V-System[THEI85], in the AT&T Bell Laboratories NEST system[AGRA85] and so on.

While all these works provided significant insight into various aspects of load sharing, the problem of communication delays and out of date state information and its impact on load sharing has not been investigated in any great detail. In this paper, we adopt the analytical approach and focus on the effect of communication delays upon the performance of simple Receiver-Initiated load sharing algorithms. In particular, our analytical models account for task transfer times and delays in acquiring remote state information.

We feel that the problem is interesting because there exist a sufficient number of system architectures that will generate significant delays in task transfers and inaccuracies in remote state information. Consequently, the question of how to deal with out of date state information has been one of the many interesting developments in designing algorithms for distributed systems as investigated in Stankovic[STAN85].

Theimer et.al[THEI85], report their concerns with task transfer delays. Furthermore, they have acknowledged that if the files used by a task were transferred (as they might have to be, if the nodes were disk-based), the effect of delays would become even more prominent (the V-System is currently comprised of diskless workstations). While network bandwidths have increased substantially in the recent past, the development of high-speed network interfaces has lagged behind considerably. This has been particularly true of microprocessor-based controllers, as reported by Lantz et.al[LANT85], which are often reported to be slower than simple controllers which use fixed logic.

In this connection, we have developed analytical models that help us better understand the effects of delays in distributed systems. Various relevant performance metrics are derived from these models and the load sharing algorithms are compared on the basis of these metrics. In a related study[MIRC87], we had determined the effects of delays upon three algorithms called Forward, Reverse and Symmetric. To provide a large breadth of study, we had made some simplifying assumptions in the analysis. For instance, K was always equal to 1 and probing times were assumed to be zero.

In this paper, we relax the simplifying assumptions made in our previous work and study Receiver-Initiated load sharing algorithms in greater depth and in a more general setting. By studying the results obtained from the model solutions, we are able to determine the exact effects of delays and out of date state information on Receiver-Initiated load sharing policies. Furthermore, we are able to determine the range of delays and traffic intensities over which state information is worth gathering and useful load sharing can be performed.

The remainder of this paper is organized as follows: In Section 2, we provide a brief description of the system architecture and the load sharing algorithms called R_K, R_{K_T}, R_D, R_{D_T} and R_{T^2}. Section 3 comprises the description of the Markov process corresponding to the R_K algorithm and its Matrix Geometric solution. The analysis corresponding to the R_{K_T}, R_D, R_{D_T} and R_{T^2} probing algorithms will only be described in brief since it is similar to the solution for R_K. In Section 4, we study the performance characteristics of the algorithms. The analytical models are validated against simulation studies. We summarize our work in Section 5. Finally, there is an appendix that describes the internals of the matrices involved with the solution of the Markov processes.

2 System Architecture and Load Sharing Algorithms

2.1 System Architecture and Motivation

In this research, we assume a network of nodes that contain the algorithms and mechanisms necessary for distributed load sharing. To implement load sharing, processing and transmission of communication messages for state updates (probes) and for task transfers can potentially generate considerable overhead at the nodes. Different system architectures can impose very different costs for these overheads. At one end of the spectrum nodes could have dedicated processors to handle communication overheads, supported by a very high bandwidth fiber-optic bus communication. On the other end, nodes could be multiplexed between application tasks and communication packet processing.

In the system that we will be considering, we have made the following assumptions. The architecture of the individual nodes includes a powerful network controller which is used to process most of the overhead generated by task and probe movement. The addition of high capability controllers to perform network related functions has been the general trend during the past few years and is likely to become even more prevelant in the future, owing to reduced costs of hardware. It is reasonable to assume that the network controllers will have a DMA capability to access main memory without much interference to the CPU. While the bulk of the overhead processing for task transfer is transferred to the controller, delays will nevertheless occur during this processing. Further, there will be network delays in the transmission of probes and tasks. We are interested in studying the combined effects of these delays.

2.2 Load Sharing Algorithms

In this subsection, we briefly describe the reverse probing algorithms that we study in this paper. Each algorithm utilizes a threshold T.

- Algorithm R_K : This reverse probing algorithm with parameter K is activated every time a task completes at a node and the total number of tasks at the node is $\leq T$. If so, the node probes a small subset of remote nodes at random to try and acquire a remote task. Only nodes that possess more than $T + 1$ tasks, (including the currently executing one) can respond positively. If more than one node respond positively, the probing node chooses one of these nodes at random from which it requests a task. Because there is a delay in acquiring a remote task, a node can request another remote task in the meantime, if a local task completes and the node is still $\leq T$. However, a node may only have a maximum of K remote tasks pending at any time. An important aspect of this study is to determine the impact of varying the parameter K.

- Algorithm R_{K_T} : This algorithm is similar to R_K, except that the node sends out reverse probes only when a task completes and the number of remaining tasks is exactly T. In this case too, a node may only have a maximum of K remote tasks pending. The reason that we are interested in studying this algorithm is because R_K has the potential to generate a large number of probes, especially if the threshold is high. In many instances, these probes are not likely to result in task transfers. Consequently, we postulate that in some situations, the extra probing of R_K may not provide any significant performance improvements over R_{K_T}.

- Algorithm R_{T^2} : Threshold based load sharing algorithms have been generally designed with one threshold. Thus, the threshold below which a node sends out reverse probes is the same as the one the probed node needs to be above, to provide a spare task. In general, this may not always be the optimal strategy, especially at high delays, where the sending threshold may have to be higher. Thus, we are interested in determining the conditions under which a dual threshold algorithm, in which the probing node utilizes a threshold T_p and the probed node uses a higher threshold T_t, may be useful.

In the above algorithms R_K, R_{K_T} and R_{T^2}, it is assumed that probes take zero time. Thus, a probing node has instantaneous knowledge about the status of the probed nodes. In general, this may not be a realistic assumption, although probes on the average experience much smaller delays than do tasks. The reason for making this assumption at this point is the resulting simplicity achieved in the analysis, because we believe that the issues of interest in the above algorithms are orthogonal to the effects of probing time. This assumption is subsequently relaxed.

To test the effect of the assumption of zero probing times, we study two algorithms R_D and R_{D_T}, corresponding to R_K and R_{K_T} respectively, except that probes experience a delay. Thus, if the node sends probes which do not result in a task transfer, this fact is made known to the probing node after an exponentially distributed time interval, with mean $1/\alpha$. Further, in R_D and R_{D_T}, we restrict the maximum number of pending remote tasks, K to exactly one, because of our results concerning the effects of parameter K (results presented in this paper). In summary,

- Algorithm R_D corresponds to R_K in the sense that upon completion of a task, a node sends out reverse probes if the number of remaining tasks at the node is $\leq T$, but negative replies to probes experience a non-zero delay.
- Algorithm R_{D_T} corresponds to the R_{K_T} algorithm in the sense that upon completion of a task, a node sends out reverse probes if and only if the number of remaining tasks at the node is exactly T, but negative replies to probes experience a non-zero delay.

3 Mathematical Analysis

In this section, we develop the analytical model for R_K. It is assumed that the task arrival process at each node is Poisson, with parameter λ. Also, the service times and task transfer times are assumed to be exponentially distributed, with means $1/\mu$ and $1/\gamma$, respectively. The task transfer time includes the time between the initiation of a transfer from a node and a successful reception of the task at the destination node. Further, we assume that the task transfer times are independent of the origin and destination of the tasks and the load placed on the network. The nodes are assumed to be homogeneous, i.e., the nodes have identical processing power and the arrival process at each node is the same. Tasks will be assumed to be executed on a First-Come-First-Served (FCFS) basis at each node.

Let $N_t^{(i)}$ be the number of tasks at node i at time t and $J_t^{(i)}$ be the probe state of node i, at time t and i be the condition code that indicates whether the node is not probing, or if it is probing, then how many remote tasks are pending. The codes are as follows:

- 0 : if not probing,
- k : if reverse probing and waiting for k tasks.

For example, in a system of M nodes, the instantaneous state of the network can be represented by the 2M-tuple

$$(N_t^{(1)}, N_t^{(2)}, ..., N_t^{(M)}; J_t^{(1)}, J_t^{(2)}, ...J_t^{(M)})$$

Due to the Poisson arrival assumption and the exponential service and task transfer times, the process corresponding to the above state description is Markovian.

It is clear that the model has a very large state space and would become difficult to solve, even for moderately sized systems. Consequently, we decompose the model such that each node can be solved independently of the others[EAGE86]. The interactions between the nodes which result in task transfers for the purpose of load sharing in the distributed system, are modelled by means of modifications to the arrival and/or departure process at each node.

We conjecture that the method of decomposition is asymptotically exact as the number of nodes tends to infinity. Actual experimental results indicate that there exists very good agreement between the model and simulations even wuen the systems are of relatively small size (= 10 nodes). Thus, the approximation is likely to be even better for larger systems. These analytical results have been validated through simulation for networks of at least 10 nodes.

The analysis of the algorithms is performed using the Matrix-geometric solution technique[NEUT81], which yields an exact solution of the model for each node. The model for the R_K algorithm will be described in detail. However, the analysis of the R_{K_T}, R_D and R_{D_T} algorithms will only be described in brief, with a presentation of the main performance metrics.

The material in this paper involves several Jacobi matrices, whose detailed definitions will be provided as described in Latouche[LATO81]. A matrix such as

$$\begin{bmatrix} b_0 & c_0 & 0 & 0 & & & \cdots & \\ a_1 & b_1 & c_1 & 0 & & & \cdots & \\ 0 & a_2 & b_2 & c_2 & & & & \\ & & & & & & & \\ \cdots & \cdots & & & a_{m-2} & b_{m-2} & c_{m-2} & 0 \\ \cdots & \cdots & & & 0 & a_{m-1} & b_{m-1} & c_{m-1} \\ \cdots & \cdots & & & 0 & 0 & a_m & b_m \end{bmatrix}$$

will be displayed as

$$\left\| \begin{array}{ccccccc} & c_0 & c_1 & \cdots & c_{m-3} & c_{m-2} & c_{m-1} \\ b_0 & b_1 & b_2 & \cdots & b_{m-2} & b_{m-1} & b_m \\ a_1 & a_2 & a_3 & \cdots & a_{m-1} & a_m & \end{array} \right\|$$

Figure 1 represents the state diagram for the R_K algorithm using an arbitrary threshold T. We define

$$y(n,j) = \lim_{t\to\infty} P(N_t = n,\ J_t = j),\ 0 \le n,\ 0 \le j \le K,$$

$$\mathbf{p_n} = (y(n,0), y(n,1), y(n,2), \ldots\ldots y(n,K)),\ 0 \le n,$$

$$\vec{\mathbf{p}} = (\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, \ldots\ldots \mathbf{p_l}, \ldots\ldots).$$

If the Markov process (N_t, J_t) is ergodic then $\vec{\mathbf{p}}$ is its steady state probability vector satisfying $\vec{\mathbf{p}}Q_K = 0$, where Q_K is the infinitesimal generator for this Markov process. Q_K has the structure of a block-tridiagonal matrix of the form

$$Q_K = \left\| \begin{array}{cccccccc} & A_0 & \dots & A_0 & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & B_{11} & A_1 & A_1 & \dots \\ B_{10} & B_{10} & \dots & B_{10} & A_2 & A_2 & A_2 & \dots \end{array} \right\|$$

where we define the matrices $B_{00}, B_{10}, B_{11}, A_2, A_1$ and A_0 in Appendix A.

In the subsequent discussion, q is the probability of failure in finding a remote task for a set of reverse probes, and $\bar{q} = 1 - q$.

When the node makes a transition below $T + 1$ on the completion of a task, it sends out reverse probes in order to get a remote task. A successful transition is represented by $\mu\bar{q}$ and an unsuccessful set of reverse probes is represented by the transition μq.

Thus, on the completion of a task when the node goes below $T + 1$, it sends out reverse probes, if it is not already waiting for K remote tasks to arrive in response to earlier reverse probes. A transition of this type is represented from (n, j) to $(n - 1, j + 1)$, where $0 < n \leq T + 1$ and $0 \leq j \leq K - 1$. The rate at which this node sends out tasks in response to nodes that asked it for tasks is μ'. Thus, the rate at which a node makes the transition (n, j) to $(n - 1, j)$, for $n \geq T + 2$ equals $\mu + \mu'$.

As can be seen from the generator Q_K, the Markov process has a regular structure comprised of the A_0, A_1 and A_2 matrices, preceded however by the irregular boundary conditions. The size of the irregular portion of the matrix depends upon the threshold at which the process is operating. For example, there will be exactly $T - 1$ columns of the matrices (A_0, B_{11}, B_{10}).

Neuts[NEUT81] examined Markov processes with such generators and determined the conditions for ergodicity. To interpret these conditions, consider the infinitesimal generator $A = A_0 + A_1 + A_2$, corresponding to the geometric part of the Markov process. One can show that A is irreducible and that there exists a unique row vector π, such that $\pi \geq 0$ and $\pi e = 1$, where e is a column vector of all ones. The stability criterion is given by

$$\pi A_2 e > \pi A_0 e.$$

Intuitively, this means that the rate of processing tasks (including the ones that are sent out of this node) is greater than the total arrival rate of tasks into this node. Thus, on the average, whenever there are more than $T + 1$ tasks at a node, the process drifts towards the boundary specified by the threshold T.

We now assume that all the values of all the parameters are known. First, the boundary conditions are determined, by solving a system of linear equations. Thus, we have

$$(\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{T+1}) \left\| \begin{array}{ccccc} & A_0 & \cdots & A_0 & A_0 \\ B_{00} & B_{11} & \cdots & B_{11} & B_{11} + RA_2 \\ B_{10} & B_{10} & \cdots & B_{10} & \end{array} \right\| = 0$$

where the number of columns in the matrix is exactly $T + 1$. We know from Neuts[NEUT81] that

$$\mathbf{p}_i = \mathbf{p}_{T+1} R^{T+1-i}, \forall i \geq T + 1$$

Thus,

$$\sum_{i \geq T+1} \mathbf{p}_i = \mathbf{p}_{T+1}(I - R)^{-1}$$

Also,

$$\sum_{i=0}^{T} \mathbf{p}_i + \mathbf{p}_{T+1}(I - R)^{-1} = 1$$

where R is the minimal solution of

$$A_0 + RA_1 + R^2 A_2 = 0$$

with $R \geq 0$ and the spectral radius of $R, sp(R) < 1$, and I is the identity matrix. The following iterative procedure is used to compute R [NELS85]:

$$R(0) = 0$$

$$R(n + 1) = -A_0 A_1^{-1} - R^2 A_2 A_1^{-1}, n \geq 0.$$

$E[N]$, the expected number of tasks at a node, and $E[D]$, the expected response time of a task, are given by the following expressions:

$$\begin{aligned} E[N] &= \sum_{i \geq 1} i\, \mathbf{p}_i\, e \\ &= \mathbf{p}_{T+1}(I - R)^{-2} + T * \mathbf{p}_{T+1}(I - R)^{-1} \\ E[D] &= \frac{(E[N] + \frac{(Flow-In)}{\gamma})}{\lambda} \end{aligned}$$

where $Flow - In$ is the flow into a node of remote tasks due to reverse probes. In the next subsection, we derive the equations required to determine the values of the unknowns q, and μ' and describe the iterative algorithm used to solve the resulting model.

3.1 Computational Procedure

Initially, it is assumed that the values for q and μ' are known and the model is solved using these values. In a typical step, a model solution is used to derive new values for q, and μ', and a new solution is computed. The iteration procedure that we use is described in a step-wise form, after the following definitions.

- $RFRO$: Flow ra.e out of tasks, as a result of reverse probes made by other nodes to this node.
- $RFRI$: Flow rate in of tasks, as a result of reverse probes made by this node.

Iteration Procedure

In the iteration procedure described below, i denotes the iteration count. $q^{(0)}, \mu'^{(0)}$ and $RFRO^{(0)}$ represent the initial values selected for the unknowns.

1. Let $i = 0$; choose values for $q^{(0)}, \mu'^{(0)}, RFRO^{(0)}$
2. Determine $Q_K^{(i)}$ from $q^{(i)}, \mu'^{(i)}$
3. Determine $R^{(i)}$
4. Solve the linear system corresponding to the boundary conditions
5. Determine $RFRO^{(i+1)}$ from the model solution
6. If $ABS(RFRO^{(i+1)} - RFRO^{(i)}) \leq \epsilon$, where ϵ is an arbitrary small number, stop, else
7. Let $i = i + 1$. Go to 2

We have observed from experiments, that the solution was insensitive to the initial values chosen for the unknown quantities. Consequently, we conjecture that there exists a unique solution to the model. Further, the number of iterations was usually small, ranging between 10 and 30.

Because of the assumption of homogeneity and because of the principle of equivalence of flow:

$$RFRO = RFRI.$$

$$RFRI = \sum_{i\geq 0} \mathbf{p}_i \, [011...1|^T \gamma,$$

where $1/\gamma$ is the mean delay in receiving a remote task. Thus, $RFRI$ denotes the total flow in due to reverse probes made by this node.

$$RFRO = \mu' \sum_{i>T+1} \mathbf{p}_i \, e$$

Thus,

$$\mu' = \frac{RFRI}{\sum_{i>T+1} \mathbf{p}_i \, e}$$

To determine q, the probability of a set of reverse probes resulting in failure, we use the following procedure:

Let

$$y = \sum_{i\leq T+1} \mathbf{p}_i \, e$$

If the node probes L_p nodes to receive a remote task, the the probability that all of them will be unsuccessful is denoted by: $q = y^{L_p}$, and $\bar{q} = 1 - q$ is the probability that at least one of the reverse probes is successful.

Algorithm R_{K_T}

Figure 2 depicts the birth-death process for the R_{K_T} algorithm, corresponding to a threshold T. Let $N_t^{(i)}$ be the number of tasks at node i at time t and $J_t^{(i)}$ be the probe state of node i, at time t and i be the condition code that indicates whether the node is not probing, or if it is probing, then how many remote tasks are pending. The codes are as follows:

- 0 : if not probing,
- k : if reverse probing and waiting for k tasks.

As can be seen from figure 2, the only time the node sends out reverse probes is when a transition is made from $(T+1, i)$ to $(T, i+1)$ on the completion of a task, $\forall i < K$. The infinitesimal generator for the Markov process corresponding to R_{K_T}, $\forall T \geq 1$, has the following form:

$$Q_{K_T} = \begin{Vmatrix} & A_0 & \dots & A_0 & A_0 & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & B_{11} & B_{11} & A_1 & A_1 & \dots \\ B_{10} & B_{10} & \dots & B_{10} & B_{20} & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

with exactly $T-1$ columns of (A_0, B_{11}, B_{10}). For $T = 0$, the generator takes the following form:

$$Q_{K_T} = \begin{Vmatrix} & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & A_1 & A_1 & \dots \\ B_{20} & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

Algorithm R_{T^2}

Figure 3 depicts the birth-death process for the R_{T^2} algorithm. As can be seen from the figure, the process has two thresholds, T_p and T_t, which represent the threshold at which probing can be performed and the threshold above which a node can transfer a task, respectively. Thus, when a task completes at a node and the remaining number of tasks is $\leq T_p$ and the node is not already waiting for a task, it sends out reverse probes. Upon receiving a probe, a node may agree to transfer a task if it possesses at least $T_t + 2$ tasks.

The infinitesimal generator for the Markov process corresponding to R_{T^2} has the following form:

$$Q_{T^2} = \begin{Vmatrix} & A_0 & \dots & A_0 & A_0 & \dots & A_0 & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & B_{11} & \dots & B_{11} & B_{11} & A_1 & A_1 & \dots \\ B_{10} & B_{10} & \dots & B_{10} & B_{20} & \dots & B_{20} & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

with $T_p - 1$ columns of A_0, B_{11}, B_{10} and $T_t - T_p$ columns of A_0, B_{11}, B_{20}. The computation procedure for this algorithm is identical to that for algorithm R_K.

3.2 Non-Zero Probing Times

Figures 4 and 5 depict the birth-death process for the R_D and R_{D_T} algorithms respectively, corresponding to a threshold T. It is assumed that the task arrival process at each node is Poisson, with parameter λ. Also, the service times and task transfer times are assumed to be exponentially distributed, with means $1/\mu$ and $1/\gamma$, respectively. In the previous algorithms R_K and R_{K_T}, negative probes took zero time. We relax this assumption in this model. The time to receive a negative reply to a set of reverse probes is exponentially distributed, with mean $1/\alpha$. It is assumed that α is independent of the number of probed nodes.

Let $N_t^{(i)}$ be the number of tasks at node i at time t and $J_t^{(i)}$ be the probe state of node i, at time t and i be the condition code that indicates whether the node is not probing, or if the node is probing and that the probe will result in a task transfer, or that the probe will not result in a task transfer. The codes are as follows:

- 0 : if not probing,
- 1 : if reverse probing and waiting for a task,
- 2 : if reverse probing and waiting for a negative reply.

The infinitesimal generator for the Markov process corresponding to the R_D algorithm has the following form:

$$Q_D = \begin{Vmatrix} & A_0 & \dots & A_0 & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & B_{11} & A_1 & A_1 & \dots \\ B_{10} & B_{10} & \dots & B_{10} & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

The infinitesimal generator for the Markov process corresponding to R_{D_T}, $\forall T \geq 1$ has the following form:

$$Q_{D_T} = \begin{Vmatrix} & A_0 & \dots & A_0 & A_0 & A_0 & A_0 & \dots \\ B_{00} & B_{11} & \dots & B_{11} & B_{11} & A_1 & A_1 & \dots \\ B_{10} & B_{10} & \dots & B_{10} & A_2 & A_2 & A_2 & \dots \end{Vmatrix}$$

with exactly $T - 1$ columns of (A_0, B_{11}, B_{10}). The form for $T = 0$ is as follows:

$$Q_{D_T} = \left\| \begin{array}{cccccc} & A_0 & A_0 & A_0 & \ldots \\ B_{00} & B_{11} & A_1 & A_1 & \ldots \\ B_{20} & A_2 & A_2 & A_2 & \ldots \end{array} \right\|$$

The computational procedure for both these algorithms is very similar to that for the R_K algorithm, which was described in detail in the previous subsection. Also, the parameters of the Markov Processes corresponding to these algorithms have the same meanings as in R_K and R_{K_T}.

Initial values of the unknown parameters are assumed to solve the model. Based upon this solution, new values of the parameters are determined. The iteration continues until the stopping criterion has been satisfied. It was seen that the iteration was insensitive to the initial values unknown parameters. Further, the number of iterations was usually small, between 10 and 20.

4 Performance Comparisons

In this section, the performance of the Receiver-Initiated load sharing algorithms will be compared to each other and to two bounds, represented by the no load balancing $M/M/1$ model for K nodes (also referred to as NLB) and the perfect load sharing with zero costs, i.e., the $M/M/K$ model. Wherever relevant, we will also compare the algorithms against a Random assignment algorithm, which transfers tasks based only upon local state information. The key performance metric for comparison is the mean response time of tasks.

A large number of parameters such as the service time, the threshold T, the probe limit L_p, the task transfer delay $1/\gamma$, the number of nodes in the network etc., can affect the performance of load sharing algorithms. In this connection, we will try to present the results that we believe are the most relevant. The presentation will be in the following sequence:

- Validation of the analytical results with simulations.
- Selection of parameter K.
- Comparison between R_K and R_{K_T}.

- Effect of non-zero negative probe times.
- Relation between response time and thresholds.
- Effects of multiple thresholds
- Network traffic density

Unless specifically mentioned otherwise, $L_p = 2$ in all the runs. Also, $S = 1/\mu$, $C = 1/\gamma$ and $A = 1/\alpha$ are the means of the service time, the task transfer delay and negative probe delay, respectively. Further, it will be assumed that $S = 1$ unit and all measurements of time will be in terms of this unit.

Validation with Simulations

We mentioned in Section 2 that the decomposition used in this paper is only an approximate solution which is conjectured to be exact for infinitely large systems. Thus, it is important to determine how well this approximation compares to simulations of finite sized systems. The simulation model consisted of 10 nodes in all cases except for $\rho = 0.9$, when the model consisted of 20 nodes. Figure 6 depicts a representative set of curves regarding this study.

The results presented here correspond to algorithm R_1 (i.e. $K = 1$), but the conclusions are generally representative of all four of the algorithms described in this paper. Because the simulation results were almost identical to the results of the analytical model, we have chosen not to depict the actual sample means of the response times from the simulations. Instead, the 95% confidence intervals of the simulation results are presented, as computed by the Student-t tests. The curves correspond to the results obtained from the analytical model. On the average, the confidence interval for the response time is less than ±.015 units about the sample mean. The only exception to this is at $\rho = 0.9$, when the confidence interval is about ±0.17 units about the sample mean.

We have observed (results not presented here) that in most of the cases, the variation between the simulation results and the analytical models is less than 2%. Furthermore, the model is invariably optimistic, compared to the simulation results. In any case, for loads ≤ 0.8, the model is a very good approximation, even for reasonably small systems. In cases where the variation is more than 2%, it was seen that by increasing the size of the simulation system to 20 nodes, the results generate better agreement with those of the analytical model. For instance, the variation at

$\rho = 0.9, C = 0.1S$, which was about 10% (results not depicted in figure), decreases to less than 5% for a system of 20 nodes.

Selection of K

In this set of tests, we determine the effects of varying K, the maximum number of pending remote tasks at a node. We would like to determine when, if at all, there is any significant gain in sending out probes while waiting for one or more remote tasks to arrive. Thus, we have compared the performance of R_1, R_2, R_3 and R_4 under a variety of conditions.

Figure 7 depicts the effect of K on optimal response times generated by various values of K for delays $0.1S, S$ and $10S$. The traffic intensity under consideration is 0.8. Also shown is the NLB response time. As can be seen, the effect of varying K is almost negligible. The best improvement over R_1 is less than 1% and this occurs at delay of S. In fact, we have observed very similar behavior for all traffic intensities less than 0.8. At $\rho = 0.9$ (results not depicted here), marginally better performance (about 4%) is exhibited by R_3 at delay = S. For delay = $10S$, the improvement is about 2.5%. The conditional mean number of pending probes (conditioned on the event that at least one probe is pending), $E[P]$, was computed for all the tests.

The insensitivity to K may be explained by the following reasons:

- At low delays, remote tasks arrive much quicker than the node is able to complete a task (and send out another probe, as a result). For $\rho = 0.8$, and delay = $0.1S$, it was seen that $E[P] = 1.0044$ $(K = 4)$, for the optimal threshold, which is zero.
- At moderate to high delays $E[P] = 1.242$ $(K = 4, Delay = S)$ and $E[P] = 1.636$ $(K = 4, Delay = 10S)$. However, the effect of the corresponding increase in load sharing is in all likelihood balanced out by the added costs due to high transfer delays.

Thus, we can conclude that the effect of K on optimal response times is negligible. In the following discussion, K will equal 1, unless mentioned otherwise.

Comparison between R_1 and R_{1_T}

The results of performance comparison between R_1 and R_{1_T} is depicted in Figures 8 and 9. The traffic intensities represented in the graphs are 0.6 and 0.9. The task transfer delay in Figure 8 is $0.1S$ while in Figure 9 it is $10S$. It is seen that at low loads, the performance of the two schemes is almost identical for the most part, with R_1 being marginally superior in some cases, especially at low delays. This can be explained by the fact that at low delays, R_{1_T} is unable to take as much advantage of the low cost of task transfer as is R_1.

For high loads, R_1 appears significantly better than R_{1_T}, for low as well as high delays. For low task transfer delays, the same argument applies as in the earlier paragraph. Thus, we will not study the performance of the threshold probing variations of the algorithms any further in this paper.

Effect of non-zero negative probe times

Figures 10 and 11 depict the effect of non-zero negative probe times on R_D. The traffic intensities under consideration are 0.5 and 0.9. The task transfer delay corresponding to the results in Figure 10 is $10S$ and in Figure 11, the delay is $0.1S$. Also shown in the figures are the baseline results corresponding to zero negative probe times.

At high transfer delays and low loads ($\rho = 0.5$), it is seen that the effect of non-zero probe times are not significant. The dominant effects are due to the task transfer delays. As a matter of fact, the response times actually become slightly worse as the negative probes arrive at higher speeds. However, for high loads, ($\rho = 0.9$), there is about 15% improvement by increasing the rate of negative responses. However, in both cases, the performance of R_D approaches that of R_1 when $A \leq 1/10D$.

The effects of non-zero probe times appear more prominent at small task transfer delays = $0.1S$, as depicted in Figure 11. This is especially evident at $\rho = 0.9$ where the slow negative probes cause a significant deterioration in performance. Here again, R_D approaches R_1 when $A \leq 1/10D$. Thus, it appears that as long as the average probing time is less than $1/10th$ the average task transfer time, the system essentially behaves like one with zero probing time. This brings us back to the argument that probes being much smaller entities than tasks (one packet of information as opposed to several hundreds or thousands in the case of tasks), it would not be unreasonable to believe that probes might take a fraction of the transfer time associated with tasks.

Response Time vs. Thresholds

Figures 12, 13 and 14 show the response times for the R_1 algorithm tested over a wide range of communication delays and thresholds, for the traffic intensities of

0.5, 0.7 and 0.9. It can be seen from Figure 12, that at low delays ($C = 0.1S$) and low to moderate loads (≤ 0.7), the optimal threshold is 0 and the performance is a monotonically increasing function of the threshold. Also, the response time generated at $T = 0$ is only about 50% better than the NLB value for moderate loads ($\rho \leq 0.7$). For example, at $\rho = 0.7$, the R_1 response time is about 1.7 units while the corresponding NLB is 3.33 units. However, at low to moderate loads, there is significant room for improvement as compared to the $M/M/K$ model, which produces a response time of about 1.04 units at $\rho = 0.7$. These observations can be explained by the following arguments:

- At low delays, the cost of transferring a task is much lower than the potential improvement due to the effect of load sharing. Thus, $T = 0$ permits very active load sharing.
- Because the delays are small, much greater certainty exists in the knowledge that an idle node will continue to remain idle during the time it takes to transfer a task to it. Thus, in some sense, $T = 0$ ensures that all task transfers are useful in that a remote task arrives at the node soon after it becomes idle.

For moderate delays ($C = S$, Figure 13), the behavior is as follows: Even at $\rho = 0.5$, there is a gain of about 15% over the corresponding NLB performance. The response time for the NLB is 2 units while the algorithm generates about 1.7 units. The improvements at higher loads are even more substantial, being as high as 66% for $\rho = 0.9$. The response time values for the algorithm and NLB in this case are 3.4 units and 10 units respectively. However, as might be expected the results do not compare very favorably against the $M/M/K$ model, which generates a response time of 1.3 units, for $\rho = 0.9$. Further, $T = 1$ for $\rho = 0.5$ and 0.7, while $T = 2$ for $\rho = 0.9$, are the *optimal* thresholds.

When the communication delays increase to the order of $10S$ (Figure 14), it is seen that the best that can be achieved for $\rho = 0.5$ is the NLB performance. Thus, it would be appropriate to turn off load sharing here. For $\rho = 0.7$, a small gain of about 5% is seen, at $T = 6$. This impovement is small enough that if the interference of probes could be accounted for, the best strategy might very likely be to turn off load sharing. However, for $\rho = 0.9$, the reduction in response time over the corresponding $M/M/1$ is about 35% (6.5 units as opposed to 10 units for NLB) and this occurs at $T = 8$.

Effects of Multiple Thresholds

To determine the usefulness of multiple thresholds, we studied the R_{T^2} algorithm over a wide range of traffic intensities and delays. The loads were varied between 0.5 and 0.9 and the delays ranged from $0.01S$ to $100S$. Initially, the optimal performance generated by algorithm R_1 (the single threshold counterpart of R_{T^2}) for the above loads and delays was recorded. The results for algorithm R_{T^2} indicated that the optimal threshold-pair (T_p, T_t) (corresponding to probing and task transfers) resided in the near neighborhood of the original threshold T (generated by R_1), as might be expected.

From the results of the model (details not presented here), it was seen that for low to moderate delays, the pair (T_p, T_t) was identical to T. This was seen to occur for all loads tested until delays of about $5S$. At delay $= 10S$, and $\rho = 0.7$, $T = 5$, the optimal performance was generated by $T_p = 5$ and $T_t = 6$. At $\rho = 0.9$ and delay $= 10S$, $T = 7$ for R_1 whereas $T_p = 7$ and $T_t = 8$ generated the optimal performance. However, the response time improvements in both these cases were almost insignificant, being less than 0.25%. As the delays were increased, the pattern was very similar, with little or no improvement being noticed over the single threshold algorithm. Thus, we can conclude that there appears to be no benefit in utilizing dual thresholds for the types of load sharing policies we have studied.

Network Traffic Density

Figure 15 depicts the effects of transfer delays on the amount of network traffic generated by the nodes running algorithm R_1 for traffic intensities 0.5, 0.7 and 0.9. For each load, two curves are presented, one depicting the rate at which a node generates probes and other the rate at which tasks are transferred, which is the same as the flow into or out of a node. These results correspond to the optimal behavior generated by this algorithm under the delays indicated. For $\rho = 0.5$ and 0.7, we can see the following behavior: The task flow rate drops to zero at about $10S$ for $\rho = 0.5$ and at about $40S$ for $\rho = 0.7$. The probe curves for these loads follow a more or less increasing function until the delays corresponding to zero task transfer rate are reached. At this point, the curves stabilize at the value corresponding to the following equation:

$$r \doteq L_p \mu \sum_{i=1}^{T+1} \bar{\mathbf{p}}_{\mathbf{i}} |10|^T,$$

where r is the probe rate for a node at a particular traffic intensity. The optimal thresholds for extremely high delays (about $40S$) and loads ≤ 0.7 are very high (in essence, they are infinite because no load sharing is being performed). Thus, every time a task completes, a node sends out reverse probes.

At first glance, the behavior for traffic intensity 0.9 appears to violate the above rules. One is inclined to believe that probe rates should increase with loads, for high delays. However, this is not borne out by the probe curve for $\rho = 0.9$. The reason for this behavior is the following: As the delays increase, the optimal performance produces very little load sharing. However, as long as the flow rate of tasks is greater than zero, a large fraction of the time is spent waiting for remote tasks to arrive and consequently the nodes do not send out probes upon completion of tasks (recall that for algorithm R_K, there can be at most K pending remote tasks. In this set of tests, we have set $K = 1$). From Figure 15, we can see that even at delay = $100S$, the flow rate of tasks is about 0.01 units. From the above equation, we can predict that at even larger delays, when load sharing at $\rho = 0.9$ will cease completely, the probe rate will stabilize at 1.8 units. Thus, Receiver- Initiated load sharing algorithms appear to have the shortcoming that even though no load sharing is performed at very high delays, the nodes continue to generate probes at a very high rate. These probes could potentially interfere with other traffic on the network. Sender-Initiated load sharing algorithms do not possess this property[TOWS87].

5 Summary and Conclusions

This study was concerned with the performance analysis of Receiver-Initiated load sharing policies in the presence of task transfer and probing delays. The algorithms that we tested were called $R_K, R_{K_T}, R_D, R_{D_T}$ and R_{T^2}. All of the above algorithms were tested over a large range of parameter values. In addition to the task transfer delays, R_D and R_{D_T} were subjected to the effects of probing delays.

The analysis of the distributed load sharing algorithms was carried out using the Matrix-Geometric solution technique. Because the Markov process of the entire network appeared to be computationally intractable, the system was solved by decomposing the state of the Markov process. This decomposition resulted in an approximate solution for the original Markov process. However, comparisons with simulation studies of 10-20 node systems indicated that the decomposition was very accurate, with the variation being less than 5%.

Some of the key observations that we made from our studies were as follows:

- In an earlier study[MIRC87], we had restricted the value of K (the maximum number of pending remote tasks at a node) to exactly one. The motivation for this was the resulting simplicity, because in the general case, the value of K could be infinite. In this paper, we studied the effect of varying K on algorithm R_K and concluded that under most values of parameters, $K = 1$ is a very good approximation for the general case.

- In comparison studies between R_1 and R_{1_T}, it was seen that when the transfer delays were small, R_1 outperformed R_{1_T} quite convincingly, over most of the range of traffic intensities tested. This effect is more prominent at high loads and low to moderate delays because more active load sharing can be performed.
- In order to simplify the analysis for the R_K and R_{K_T} algorithms, we had assumed that probes took zero time, while tasks were subject to transfer delays. Subsequently in this study, we relaxed this assumption and determined the effect of probing delays. In the context of reverse probing schemes, this meant that negative replies to probes took non-zero times. In all the studies that we conducted for this aspect of the problem, it was seen that as long as probing time was less than $1/10th$ of the task transfer times, the system essentially behaved like one with zero probing times. We postulate that that probing will take a small fraction of the task transfer time, because of the possible relative sizes of these two entities. Thus, it seems reasonable to assume that probes take zero time.
- A representative study of algorithm R_1 over a large range of loads and delays was performed. It was seen that the most significant gain in performance over NLB was seen at high loads (≥ 0.8). At low to moderate delays, load sharing was viable even for low loads. At very high loads, $\rho \geq 0.9$ there appeared to be a substantial benefit from load sharing, even when delays were very high, as much as $10S$.
- We studied the effects of dual thresholds on an algorithm called R_{T^2} and noticed that very little or no benefits accrued from this change.
- We studied the effects of delays on network traffic densities. It was seen that as delays increase, the optimal performance generates fewer and fewer task transfers, with complete cessation of load sharing when delays become very high. However, nodes continue to generate probes at a fairly high rate (which stabilizes after a specific value of delay which is dependent upon load). This would appear to be a shortcoming of Receiver-Initiated load sharing algorithms.

Appendix A

We now give closed form representations of the matrices A_0, A_1, A_2 and the boundary matrices for the $R_K, R_{K_T}, R_{T^2}, R_D$ and R_{D_T} algorithms. For ease of representation, we have assumed $K = 1$.

To determine q, the probability of a set of reverse probes resulting in failure. we use the following procedure:

Let

$$y = \sum_{i \leq T+1} p_i \, e$$

If the node probes L_p nodes to receive a remote task, the the probability that all of them will be unsuccessful is denoted by: $q = y^{L_p}$, and $\bar{q} = 1 - q$ is the probability that at least one of the reverse probes is successful.

Algorithm R_K

$$B_{00} = \begin{bmatrix} -\lambda & 0 \\ 0 & -(\lambda + \gamma) \end{bmatrix}$$

$$B_{10} = \begin{bmatrix} \mu q & \mu \bar{q} \\ 0 & \mu \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} -(\mu + \lambda) & 0 \\ 0 & -(\mu + \gamma + \lambda) \end{bmatrix}$$

$$A_0 = \begin{bmatrix} \lambda & 0 \\ \gamma & \lambda \end{bmatrix}$$

$$A_1 = \begin{bmatrix} \delta & 0 \\ 0 & -\gamma + \delta \end{bmatrix}$$

$$A_2 = (\mu + \mu^{'}) I_2,$$

where I_2 is the identity matrix of size 2 and $\delta = -(\mu + \mu' + \lambda)$

Algorithm R_{K_T}

The internals of A_0, A_1 and A_2 for this algorithm are identical to those of the R_K algorithm. This is obvious because the two Markov processes have identical structures after $T + 1$. Further, the B_{00} matrices are also identical. However,

$$B_{10} = \begin{bmatrix} \mu & 0 \\ 0 & \mu \end{bmatrix}$$

$$B_{20} = \begin{bmatrix} \mu q & \mu \bar{q} \\ 0 & \mu \end{bmatrix}$$

Algorithm R_{T^2}

The internals of the matrices $A_0, A_1, A_2, B_{00}, B_{10}$ and B_{11} for this process are identical to those for algorithm R_K. However, there is the matrix B_{20} which does not have a counterpart in R_K.

$$B_{20} = \begin{bmatrix} \mu & 0 \\ 0 & \mu \end{bmatrix}$$

Algorithm R_D

$$B_{00} = \begin{bmatrix} -\lambda & 0 & 0 \\ 0 & -(\gamma + \lambda) & 0 \\ \alpha & 0 & -(\alpha + \lambda) \end{bmatrix}$$

$$B_{10} = \begin{bmatrix} 0 & \mu \bar{q} & \mu q \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} \sigma & 0 & 0 \\ 0 & -\gamma + \sigma & 0 \\ \alpha & 0 & -\alpha + \sigma \end{bmatrix}$$

$$A_0 = \begin{bmatrix} \lambda & 0 & 0 \\ \gamma & \lambda & 0 \\ \alpha & 0 & \lambda \end{bmatrix}$$

$$A_1 = \begin{bmatrix} \delta & 0 & 0 \\ 0 & -\gamma + \delta & 0 \\ 0 & 0 & -\alpha + \delta \end{bmatrix}$$

$$A_2 = (\mu + \mu')I_3,$$

where I_3 is the identity matrix of size 3 and $\sigma = -(\mu + \lambda)$.

Algorithm R_{D_T}

The matrices A_0, A_1, A_2 and B_{11} for this algorithm are the same as the corresponding ones for algorithm R_D. However,

$$B_{20} = \begin{bmatrix} 0 & \mu\bar{q} & \mu q \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

$$B_{10} = \begin{bmatrix} \mu & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{bmatrix}$$

References

[AGRA85] Agrawal, R. and A. Ezzat, "Processor Sharing in NEST: A Network of Computer Workstations," *Proc. 1st Int'l Conf. on Computer Workstations*, Nov. 1985.

[BOKH79] Bokhari, S., "Dual Processor Scheduling With Dynamic Reassignment," *IEEE Trans. Soft. Engg.*, Vol. SE-5, No. 4, July 1979.

[BRYA81] Bryant, R. and R. A. Finkel, "A Stable distributed scheduling algorithm," *Proc 2nd Intl. Conf. Dist. Comp. Syst.*, April 1981.

[dSeS84] de Souza e Silva, E. and M. Gerla, "Load Balancing in Distributed Systems With Multiple Classes and Site Constraints," *Performance '84*, , pp. 17–33, 1984.

[EAGE86] Eager, D., E. Lazowska, and J.Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Soft. Engg.*, Vol. SE-12, No. 5, pp. 662–675, May 1986.

[LANT85] Lantz, K. A., W. I. Nowicki, and M. M. Theimer, "An Empirical Study of Distributed Application Performance," *IEEE Transactions on Software Engineering*, Vol. SE-11, October 1985.

[LATO81] Latouche, G., "Algorithmic Analysis of a Multiprogramming-Multiprocessor Computer System," *J. ACM*, Vol. 28, October 1981.

[LEE87] Lee, K. J., *Load Balancing in Distributed Systems*, PhD thesis, ECE Dept., University of Massachusetts, February 1987.

[LIVN82] Livny, M. and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Performance Evaluation Review*, Vol. 11, No. 1, pp. 47–55, 1982.

[MIRC87] Mirchandaney, R., D. Towsley, and J. A. Stankovic, "Analysis of the Effects of Delays on Load Sharing," *IEEE Transactions on Computers*, 1987. submitted for review.

[NELS85] Nelson, R. and B. R. Iyer, "Analysis of a Replicated Data Base," *Performance Evaluation*, Vol. 5, pp. 133–48, 1985.

[NEUT81] Neuts, M., *Matrix-Geometric solutions in Stochastic Models: An Algorithmic Approach, Mathematical Sciences*, Johns Hopkins University Press, 1981.

[STAN84] Stankovic, J., "Simulations of Three Adaptive Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks*, Vol. 8, No. 3, pp. 199–217, June 1984.

[STAN85] Stankovic, J., "Bayesian Decision Theory and Its Application to Decentralized Control of Task Scheduling," *IEEE Transactions on Computers*, Vol. C-34, No. 2, pp. 117–130, February 1985.

[STON78a] Stone, H., "Critical Load Factors in Two Processor Distributed Systems," *IEEE Trans. Soft. Engg.*, Vol. SE-4, No. 3, May 1978.

[STON78b] Stone, H., "Multiprocessor Scheduling with the Aid of Network Flow algorithms," *IEEE Trans. Soft. Engg.*, Vol. SE-3, No. 1, May 1978.

[TANT85] Tantawi, A. and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *J. ACM*, Vol. 32, pp. 445–465, Apr. 1985.

[THEI85] Theimer, M.. K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings of the 10th Symposium on Operating System Principles*, December 1985.

[TOWS86] Towsley, D., "The Allocation of Programs Containing Loops and Branches on a Multiple Processor System," *IEEE Trans. Soft. Engg.*, Vol. SE-12, pp. 1018–1024, October 1986.

[TOWS87] Towsley, D. and R. Mirchandaney, "The Effects of Communication Delays on the Performance of Load Balancing Policies in Distributed Systems," *Proc. 2nd. Intl. Workshop on Appl. Math. and Perf./Rel. Models in Computer/Communications*, May 1987.

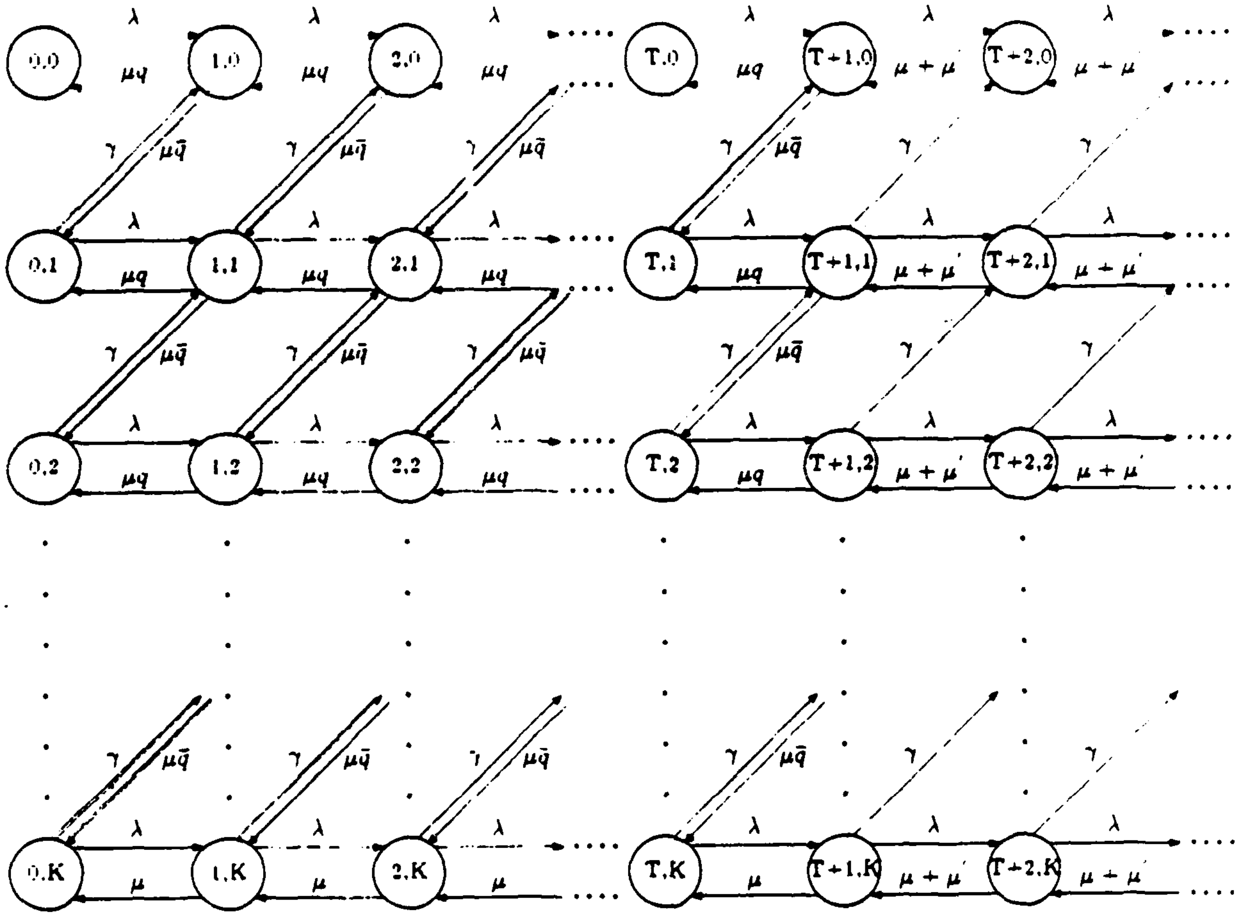


Figure 1. State Diagram for Algorithm R_K

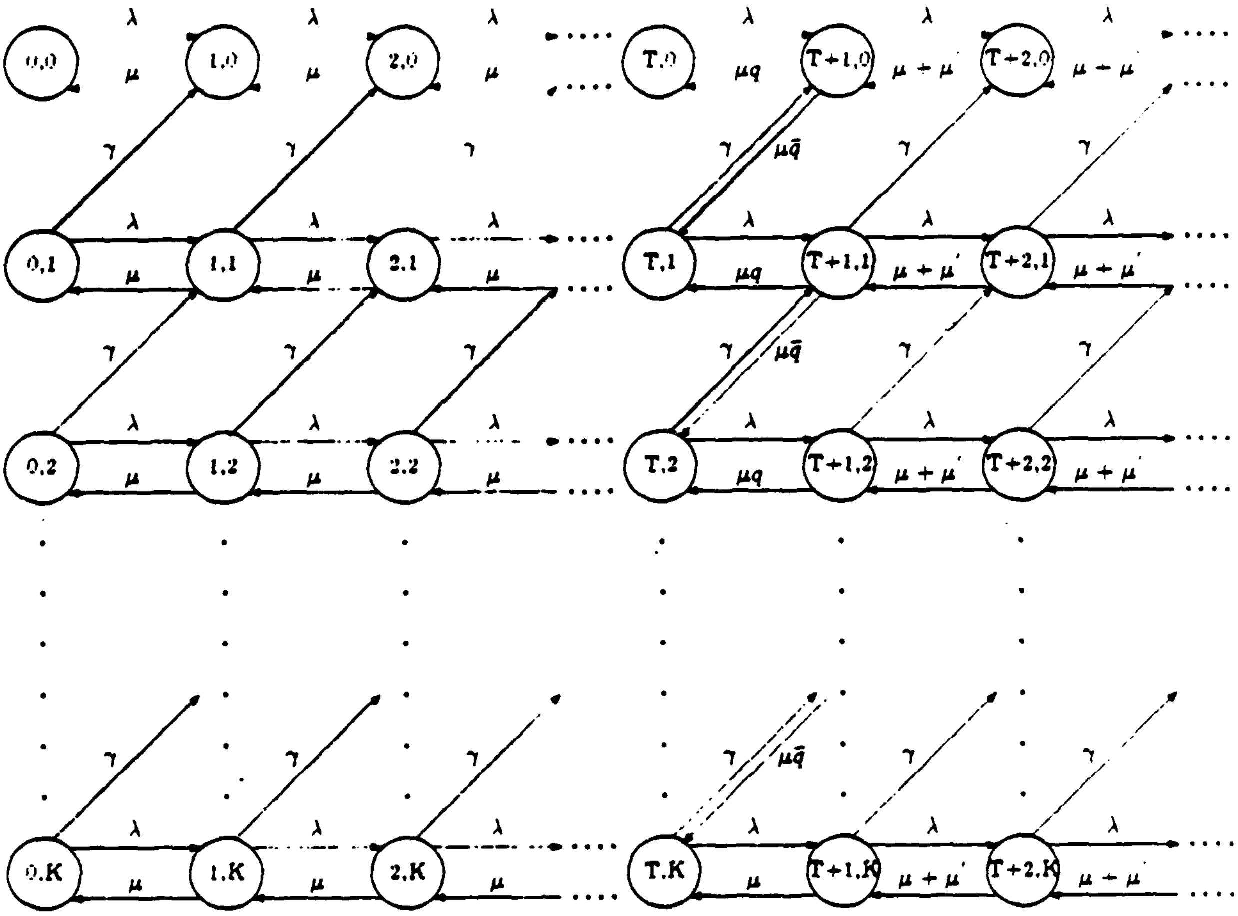


Figure 2. State Diagram for Algorithm R_{K_T}

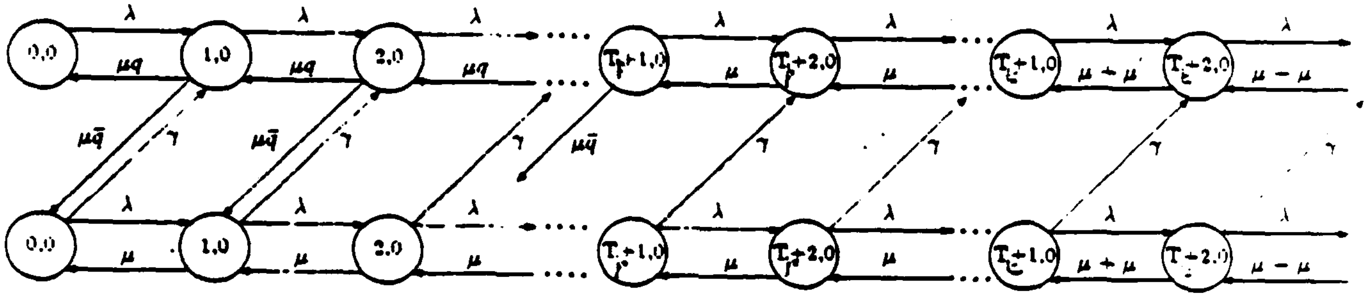


Figure 3. State Diagram For Algorithm R_{T}:

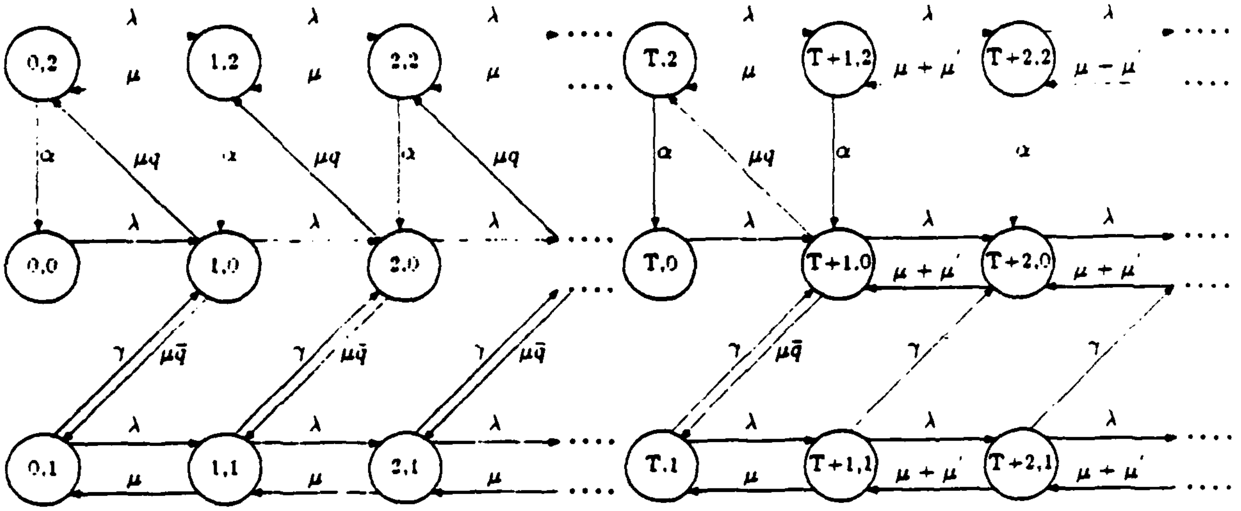


Figure 4. State Diagram for Algorithm R_D

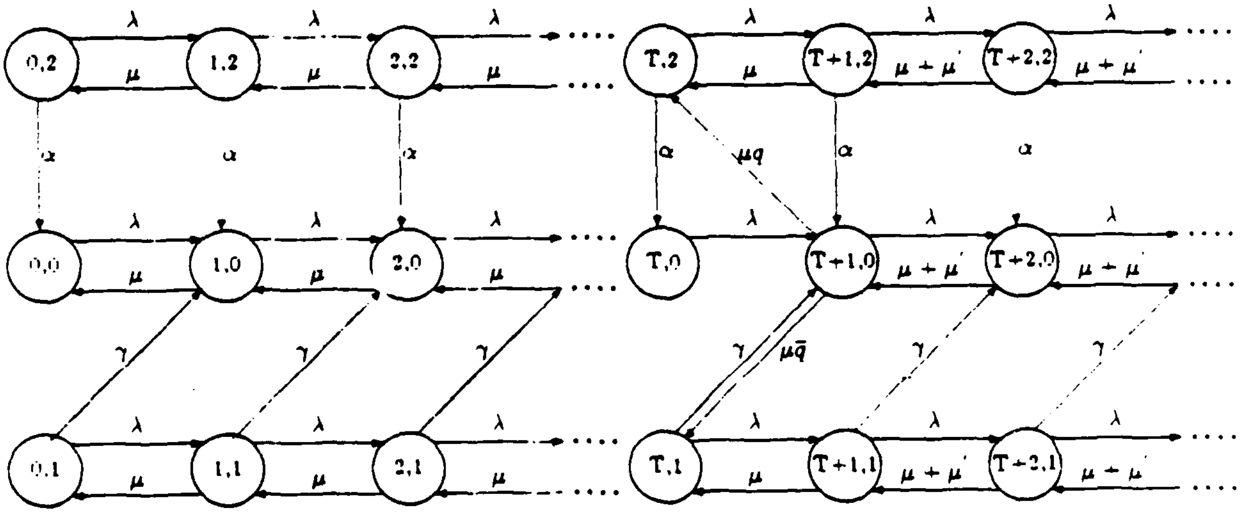
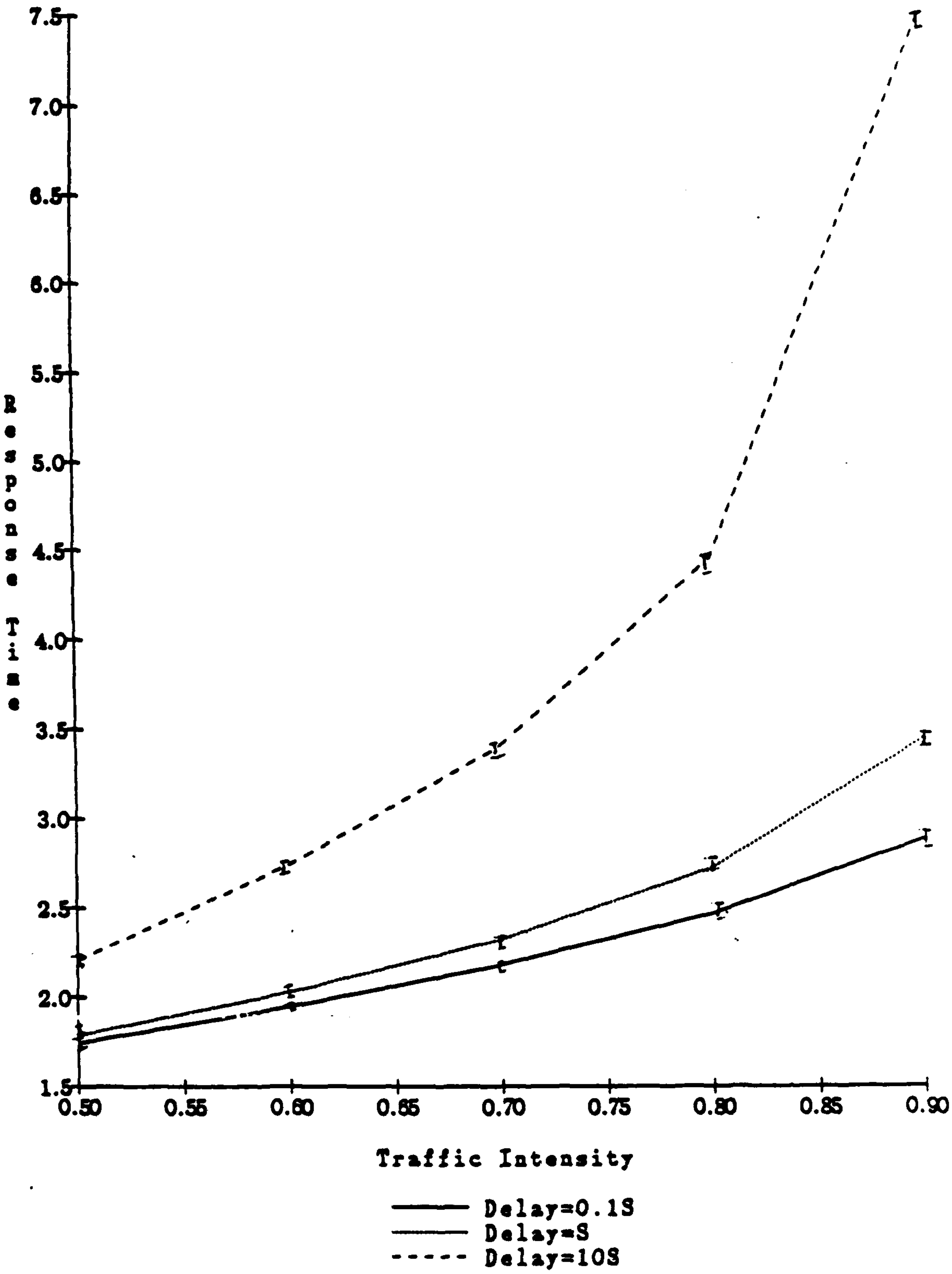


Figure 5. State Diagram for Algorithm R_{D_T}

Figure 6. Validation With Simulations



I - Confidence Intervals

Figure 7. Optimal R.T. vs. K (load=0.8)

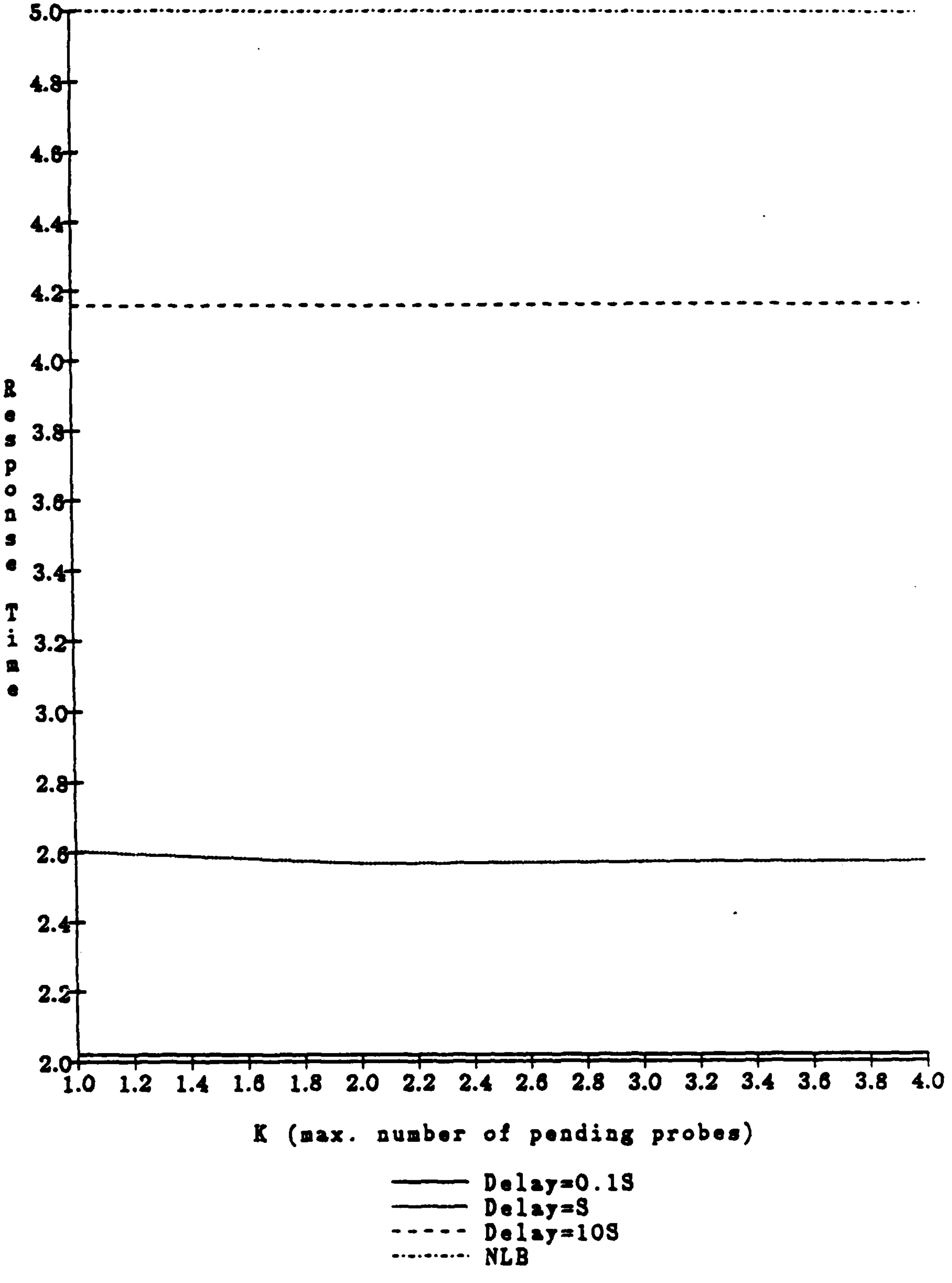


Figure 8. R1 vs. R1T (Delay=0.1S)

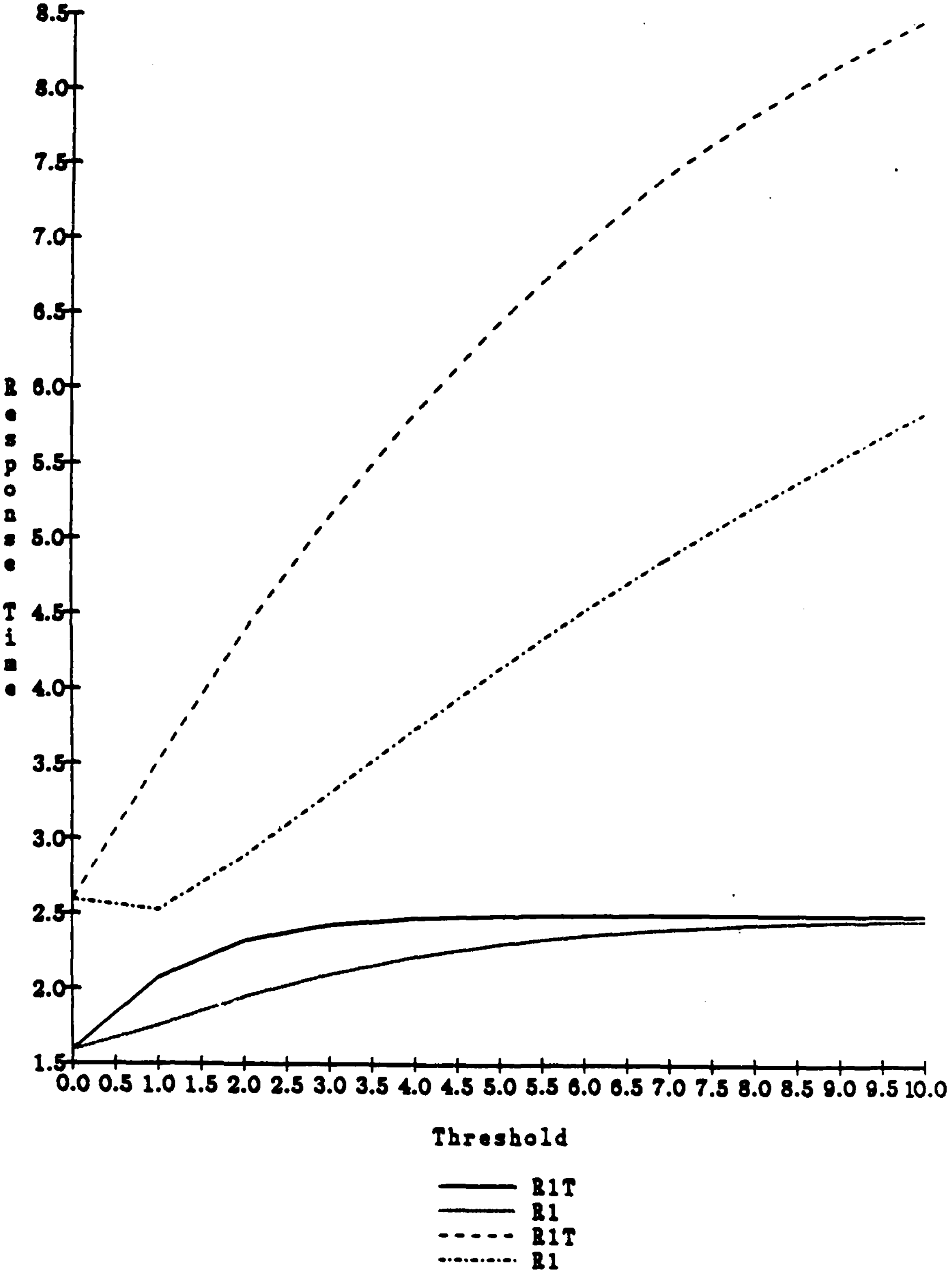


Figure 9. R1 vs. R1T (Delay=10S)

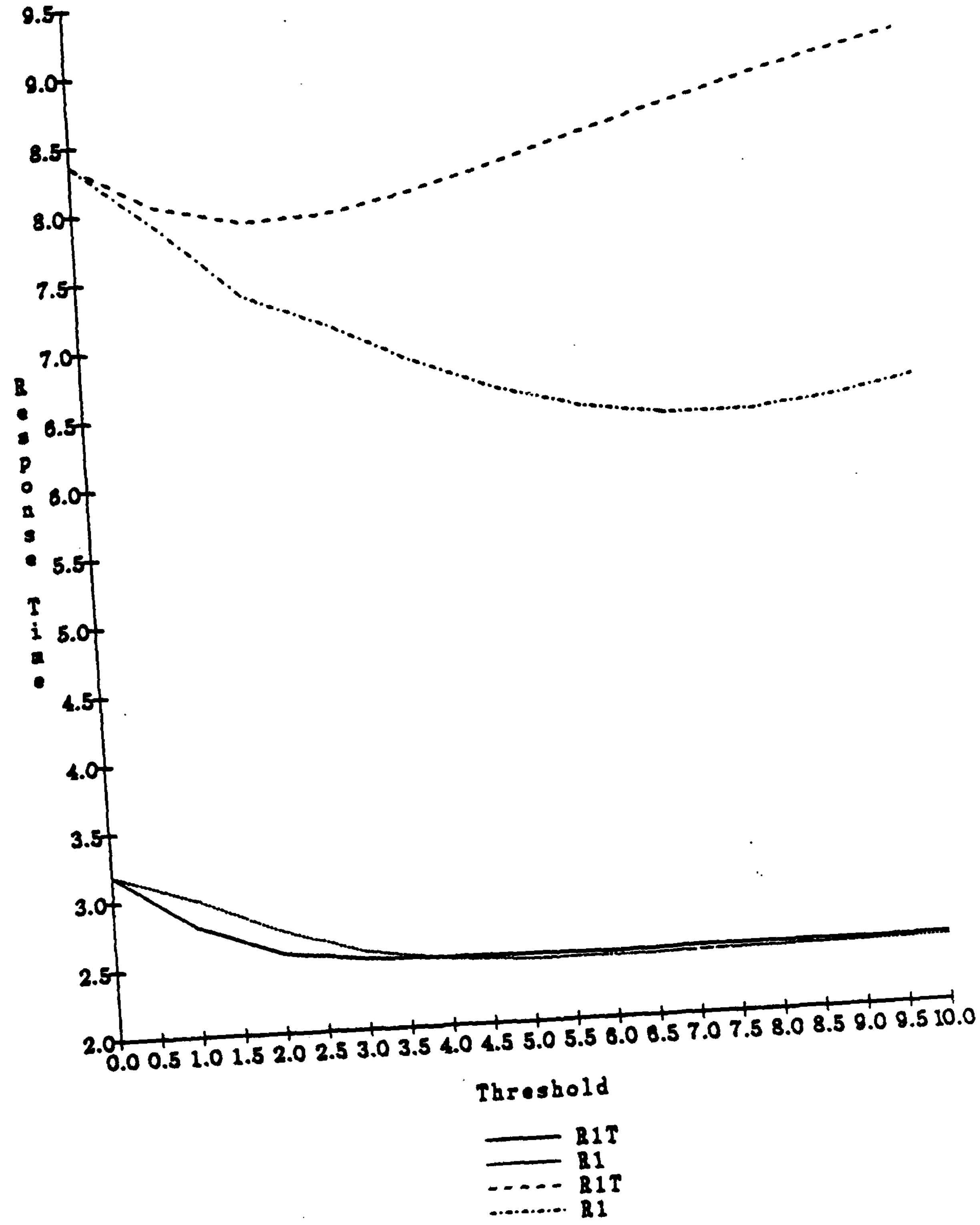


Figure 10. Effect of Non-Zero Probe Times(Delay=10S)

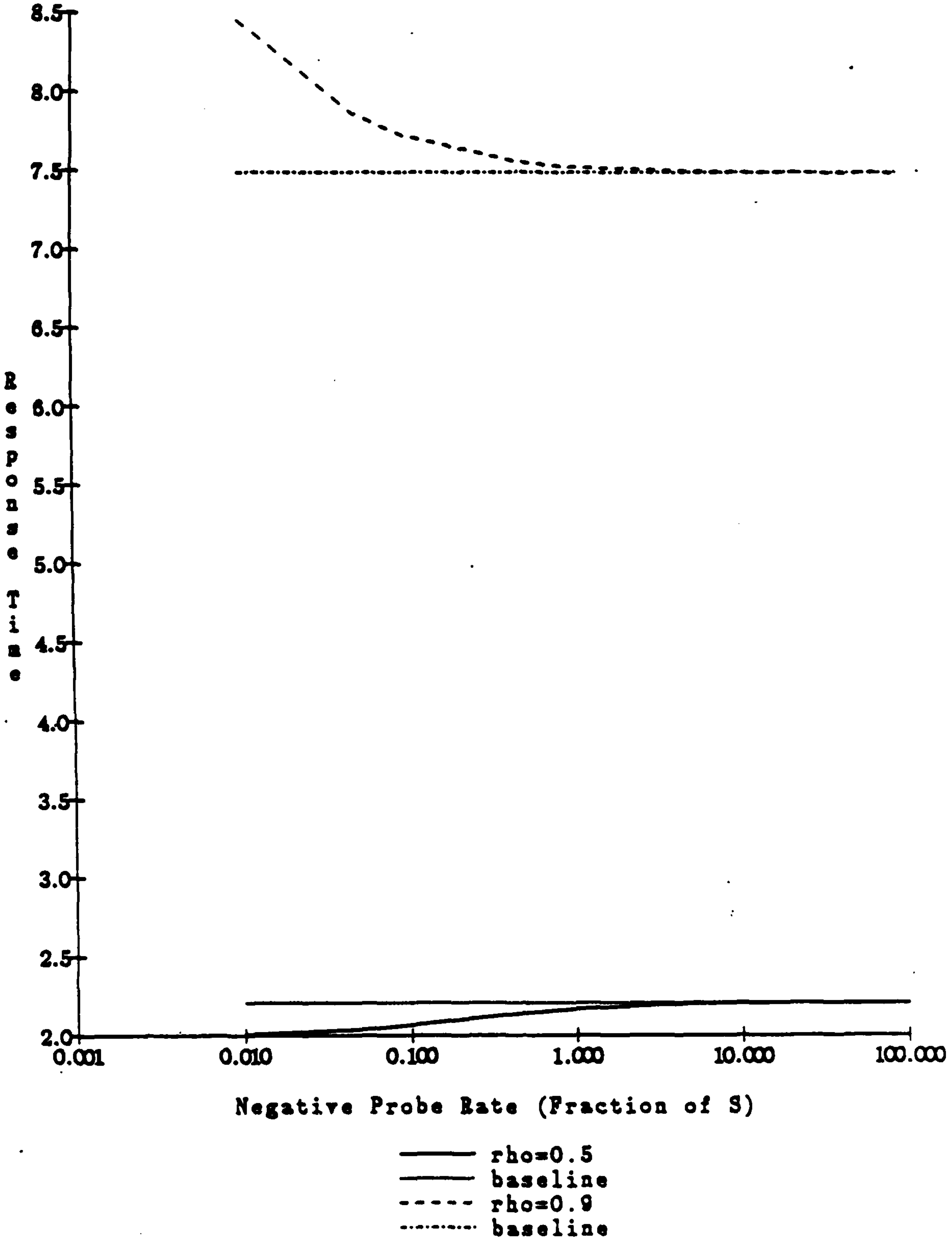


Figure 11. Effect of Non-Zero Probe Times(Delay=0.1S)

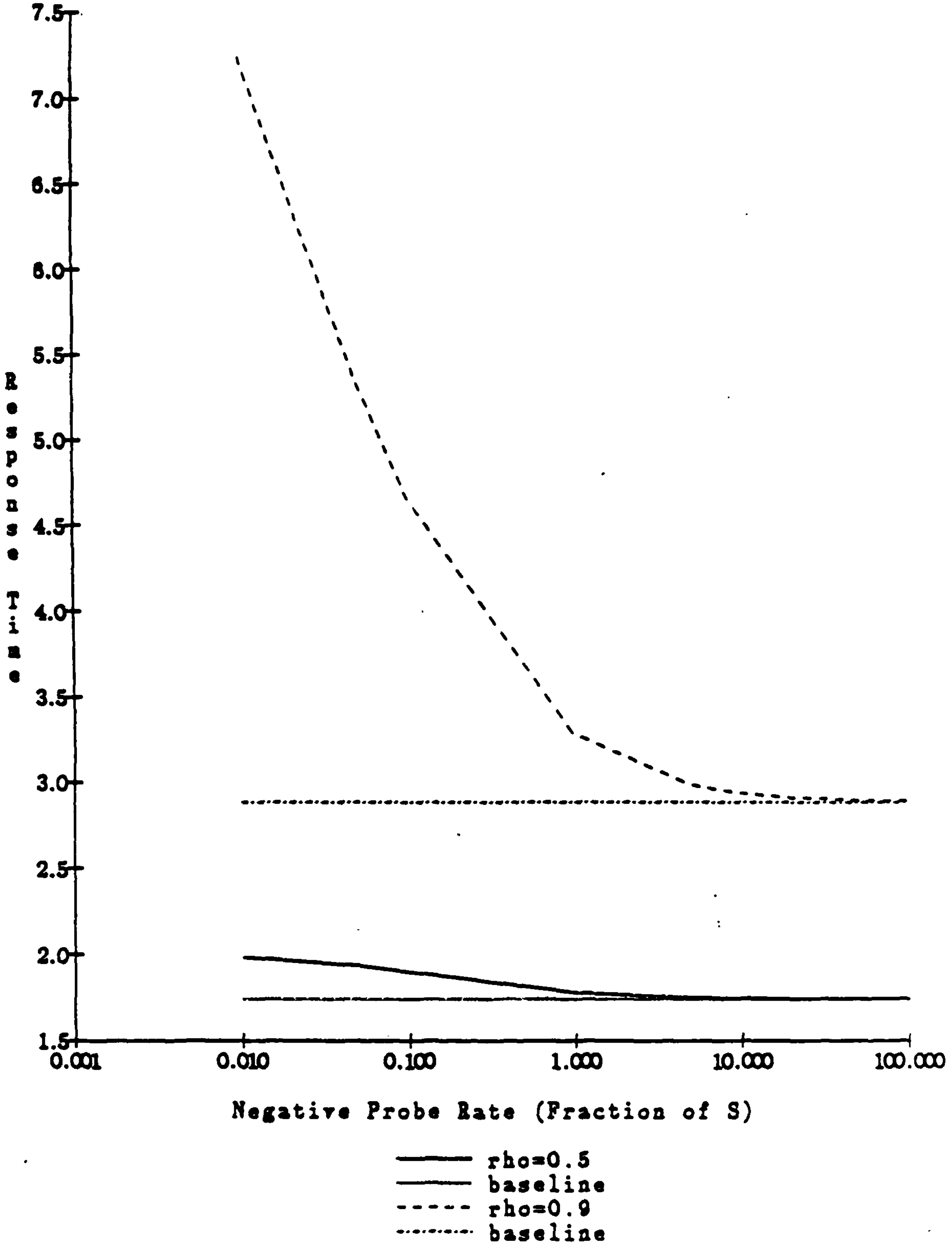


Figure 12. R.T. vs. Thresholds (Delay=0.1S)

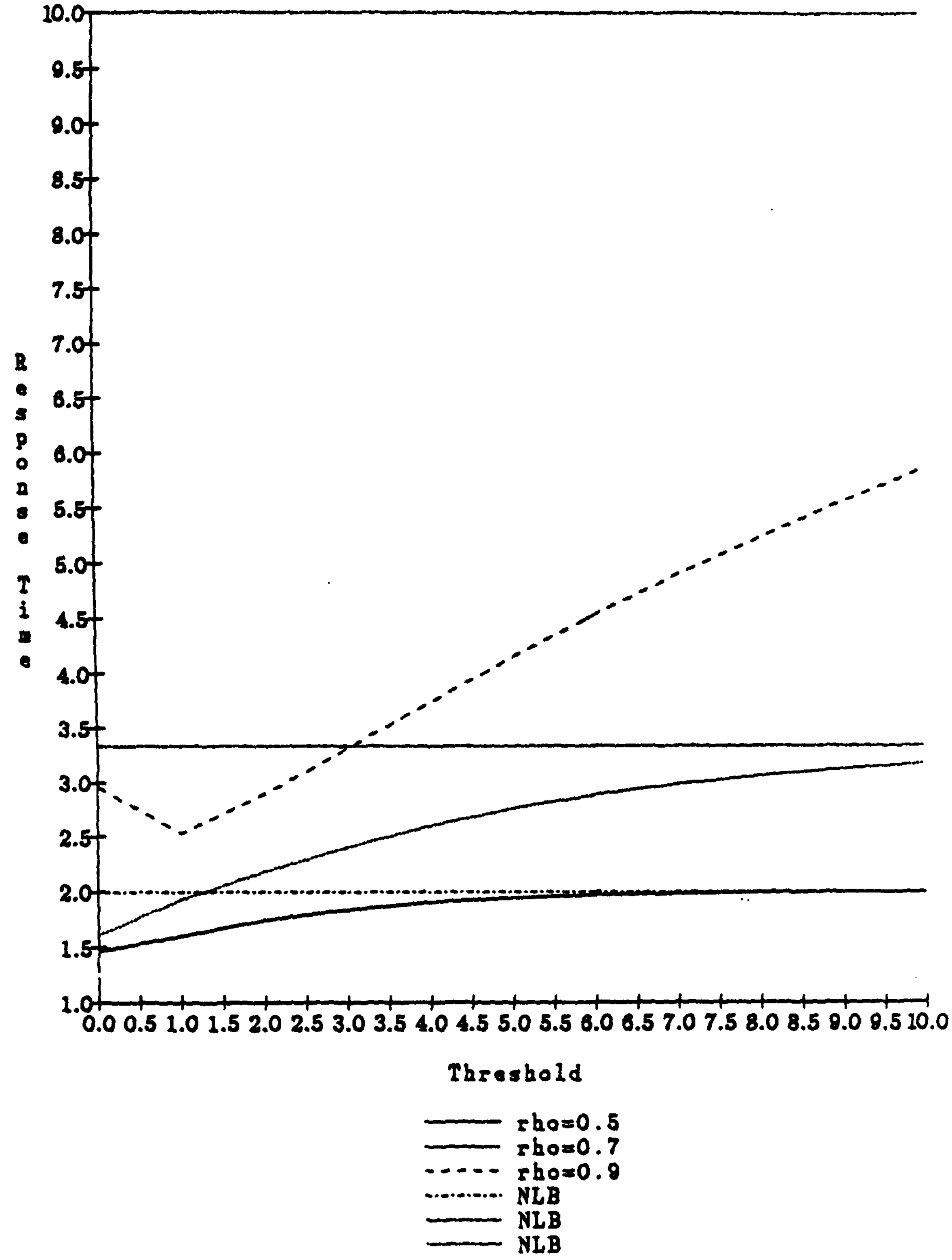


Figure 13. R.T. vs. Thresholds (Delay=S)

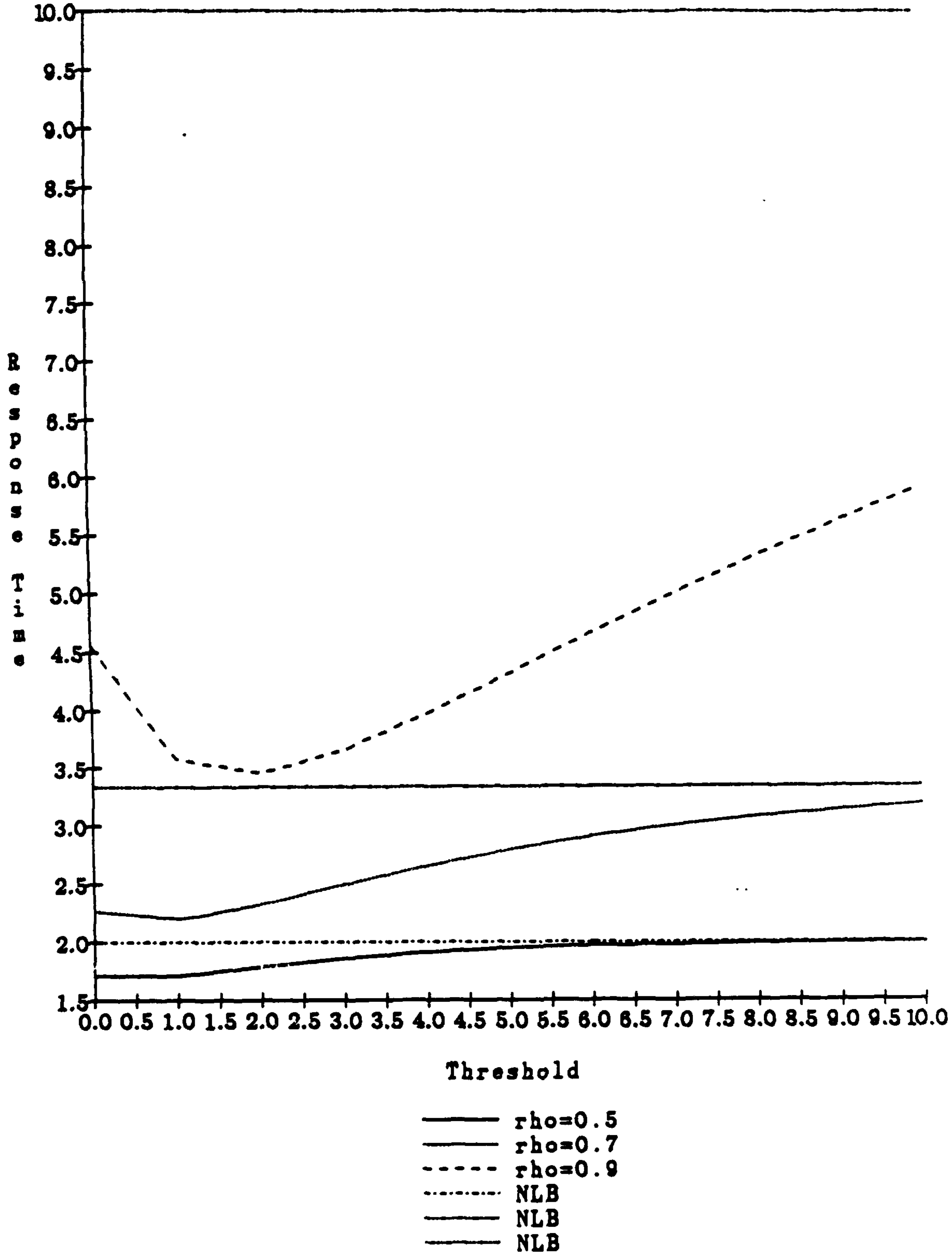


Figure 14. R.T. vs. Thresholds (Delay=10S)

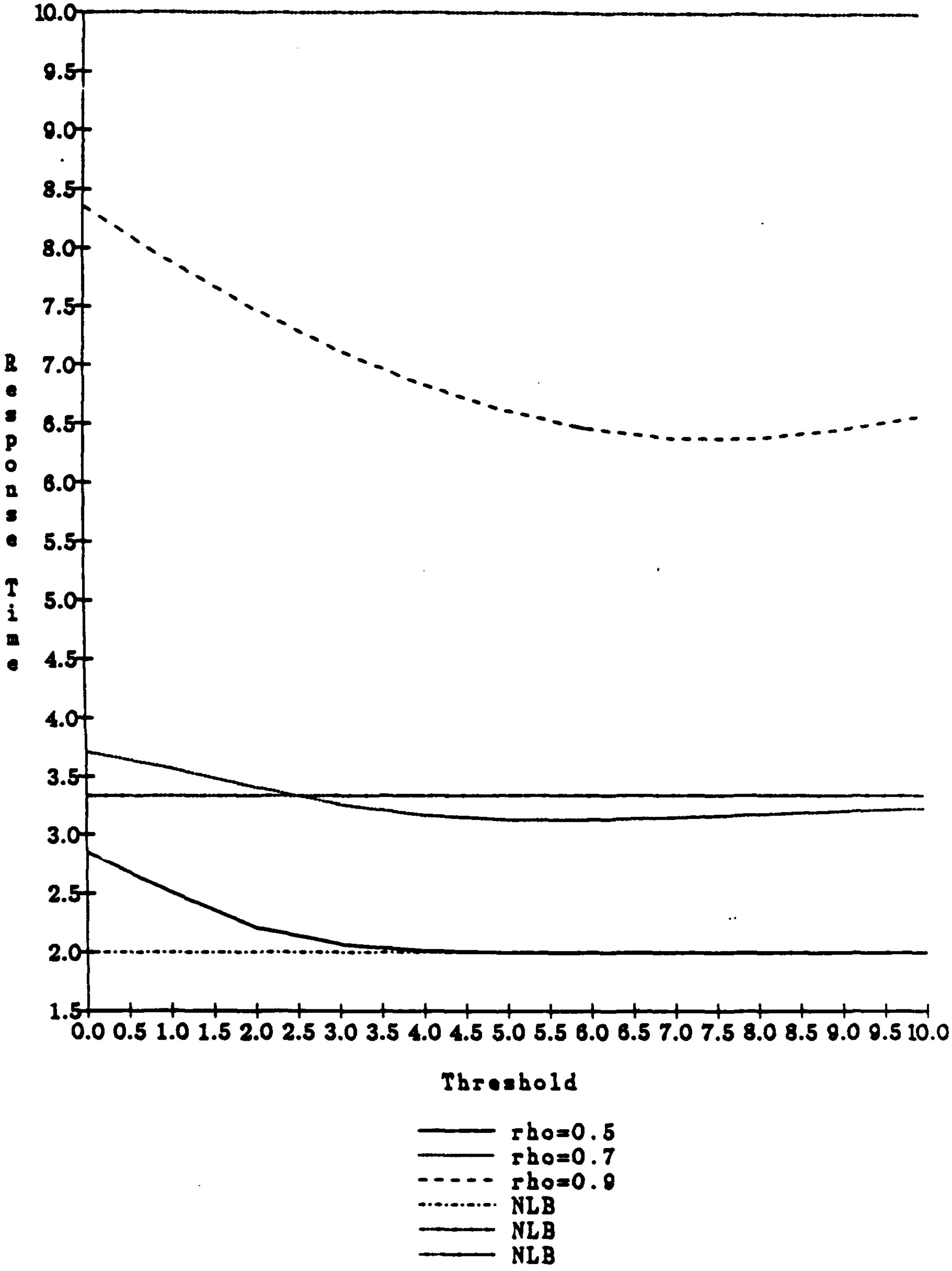
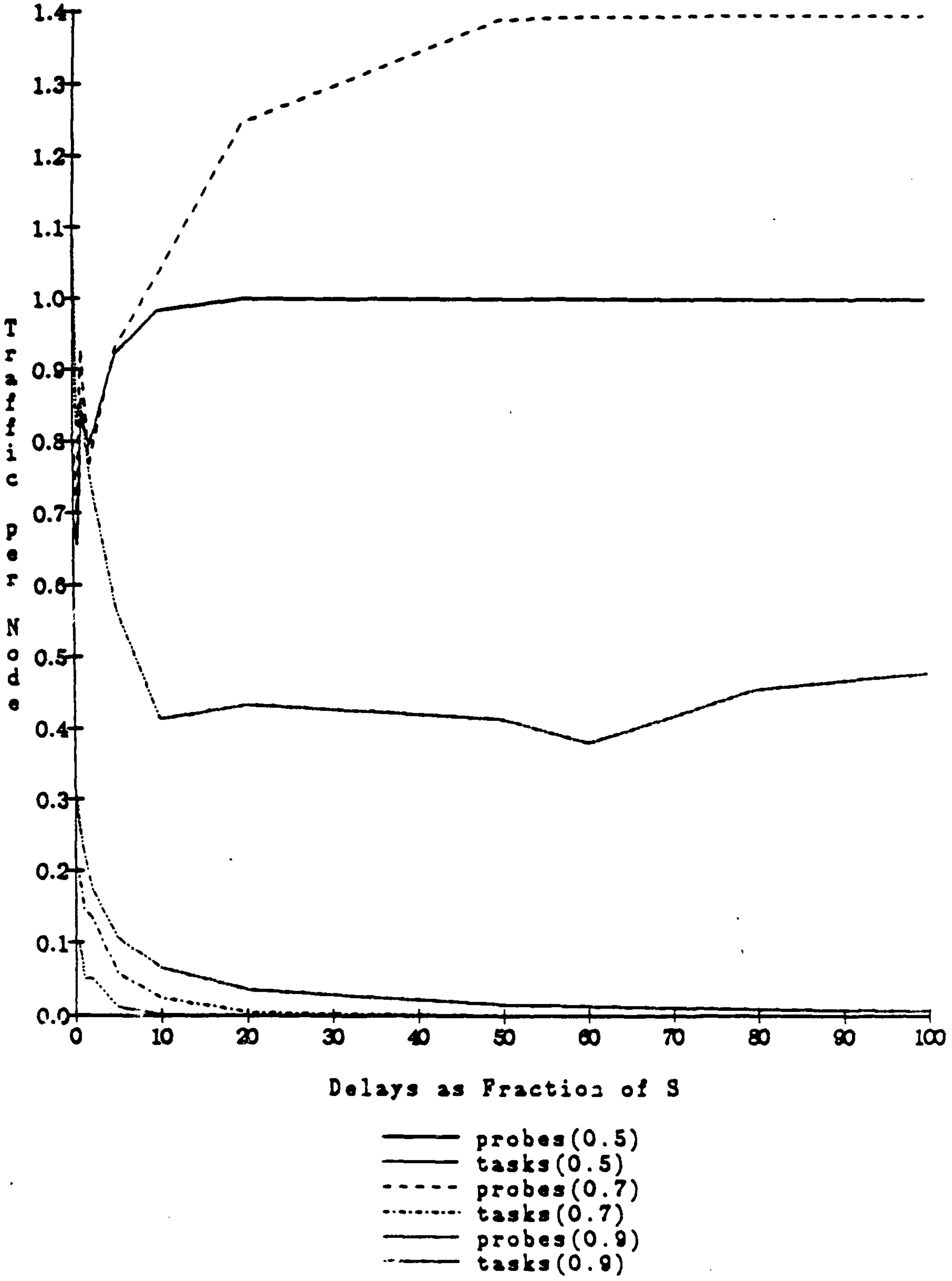


Figure 15. Network Traffic Rates


Traffic per Node
Delays as Fractio: of S
probes(0.5)
tasks(0.5)
probes(0.7)
tasks(0.7)
probes(0.9)
tasks(0.9)